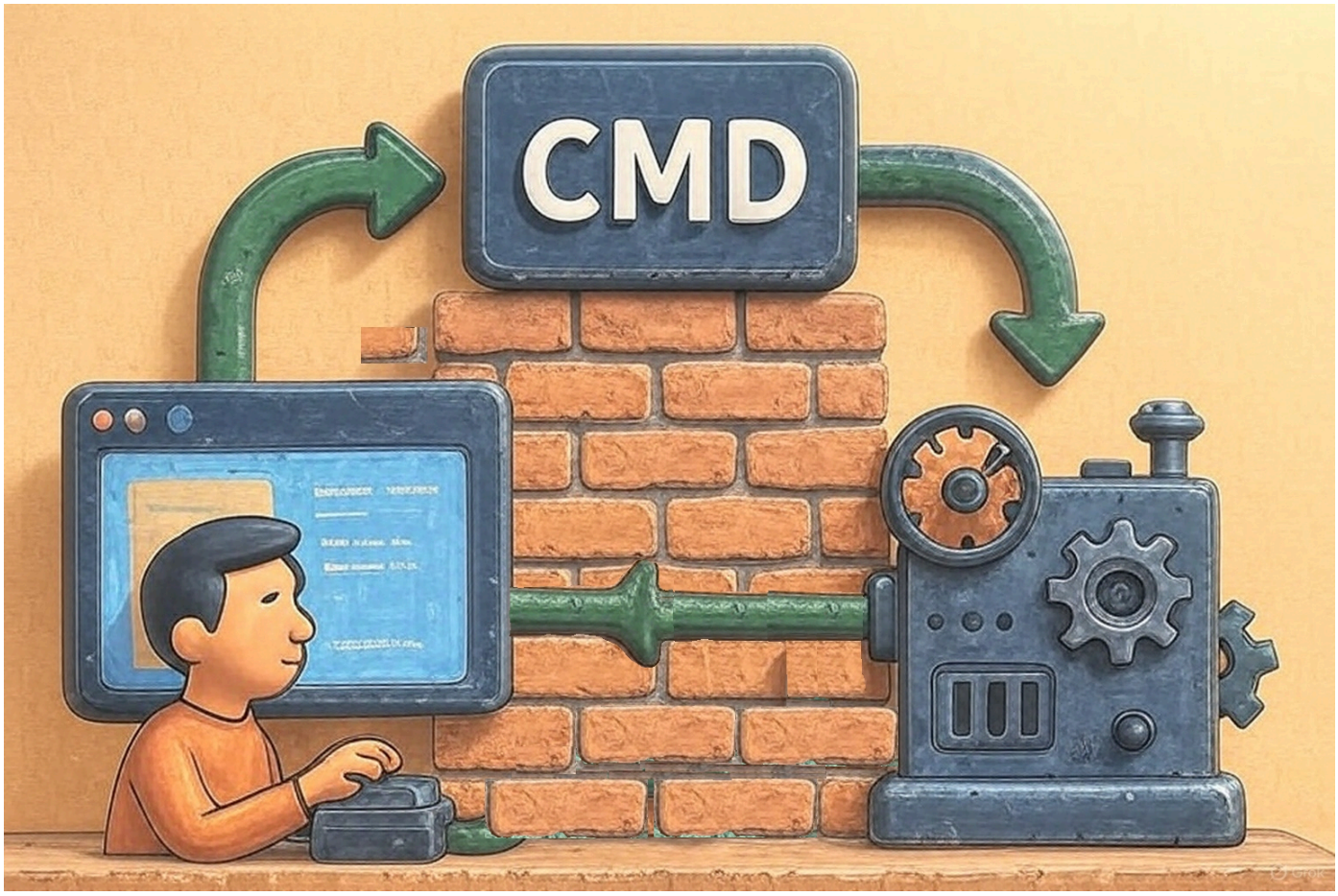
Command Pattern

Petr Talla
petr.talla@gmail.com

20-Sep-2025

Introduction



This is the most useful and at the same time perhaps the most poorly interpreted pattern. It made it into the pattern bible (Design Patterns: Elements of Reusable Object-Oriented Software by E. Gamma, R. Helm, R. Johnson, J. Vlissides) in a crippled form and has been traveling the internet that way. This creates informational noise from which its essential substance can't recover.

```
class Command {
public:
    virtual Command();
    virtual void Execute() = 0;
protected:
    Command();
};
```

That's the design from the book. And that's where everything goes wrong. It's not possible to have an effective command without arguments. Then there's this example from the same book:

```
class OpenCommand : public Command {
public:
    OpenCommand(Application*);
    virtual void Execute();
protected:
    virtual const char* AskUser();
```

```
private:
    Application* _application;
    char* _response;
};
```

You have to open some sort of dialog from the application that supplies parameters. Why? We can simply call `FindCommand("OpenCommand").Execute("C:docs/test.txt")`. File selection is part of the UI; no application needs to handle that.

Similarly, take a command like `SetColor`. Should the application provide a dialog to choose a color? And what else? That's UI work again.

The correct form of the command is: `FindCommand("SelectColor").Execute("200 200 200")`. It perfectly sets the active color of application. No dialog is needed to be activated.

And so on for everything else. Without arguments it's chaos.

Correction

Take the areas, where commands are traditionally used - command-line of operating systems, CAD software, and SQL.

The commands are not there `cat`, `select_color` or `SELECT`, but `cat FILE_NAME`, `select_color yellow`, `SELECT * FROM employees`.

Right, The canonical pattern is missing parameters in Execute.

So, the proper implementation of a command should look like this:

```
class Command {
public:
    virtual Command();
    virtual void Execute(Args args) = 0;
protected:
    Command();
};
```

`Execute` receives arguments. That's the whole essence of the problem the book ignored, and the reason why such a big misunderstanding about this pattern arose.

In practice, a command doesn't need to be a class at all—it can be a function. It's much more practical to write functions, which can all live in one file, rather than creating the same number of objects, each usually placed in its own file.

Moreover, a class with just a single `Execute(Args args)` method doesn't really make sense. Whatever would normally go into the constructor can instead be added to arguments, so the class truly loses its purpose. Choosing functions instead of classes is simply a practical solution. Compared to the class-based approach above, the principle remains identical.

The functions are then registered into a map, which gives us an implementation of FindCommand("OpenCommand"). Otherwise, we would just be registering objects. Typically, this registration is done in main().

So, if our program consistently uses commands, this registration effectively links the concrete application. It's possible to exclude any number of commands from the registration, or add new ones, and the application will always build without issues—just slightly different each time.

Simplification

We can then write a command as a function in the following form:

```
command_name arg1 arg2 arg3 ... --named_arg1 value1 --named_arg2 value2 ...
```

That means:

- `command_name` the name of the command, e.g. ls, grep, cat, etc.
- `arg1 arg2 arg3 ...` positional arguments that are not predefined, and whose meaning depends on the command. For example, with ls dir1 dir2, dir1 and dir2 are directories you want to list.
- `--named_arg1 value1` named arguments (with or without a value).

Examples:

```
set_line_color 200 200 200
```

or:

```
set_line_properties --color 200 200 200
```

This is essentially the same style of commands used in Unix. That's where the principle has survived, and it remains one of the system's strongest features.

It also survived in CAD/GIS systems, where I first encountered it. It was traditionally used there, likely because GUI controls consumed too much space on the small screens of the time.

The second question is: why hasn't this approach spread to other types of software?

The likely culprit is the broken canonized form of the pattern, which offers no practical benefit.

Another factor is that many programmers dislike embedding what looks like a primitive scripting language into the environment they are working in. But they gladly slip similar SQL strings into the very same program, or just as foolishly call the system—yet their own program is never afforded that luxury.

Example Program

```
open_file FILE_NAME
save_file FILE_NAME
mode_place_line
mode_delete
mode_zoom_out
mode_pan
mode_fit
xy X Y
```

Done. That's all.
If you implement these commands, you have a working program that can draw a line, save and load a document, and allow basic window manipulation.

The meaning is straightforward: the `mode_xxx` commands enable a mode that determines what happens when you click in the window and how that click is interpreted - command `xy X Y`.

A click can be triggered by the `xy` command, or alternatively the mouse button can be used to issue the same command.

Of course, it's necessary to write a UI that activates these commands. A command line is enough, but it's relatively simple to write a UI that uses them—clicking in the window would simply trigger the corresponding command.

As for the set of commands: they are only for creating and manipulating (including some UI elements manipulation). They are not for displaying. For displaying, reactive patterns are used—the created or modified object is automatically reflected in the UI. Commands have no role there.
(There is an exception: testing. For initial tests or verification, writing commands is fine—it's quick and unambiguous.)

It is true, however, that a command can serve to initialize a refresh, especially for dialogs. If your software can only be changed through commands, it's enough to perform a refresh after each command. Given the speed at which a human issues commands, this usually means once every few seconds. For command batches we can block refres for each command separatelly.

For more sophisticated refreshing, we can use a reactive pattern. In practice, though, the two approaches won't differ much—under a reactive model, a command triggers an object change, and that triggers the refresh.

The advantage of the reactive model is that updates are limited to what has actually changed. In a UI, however, such fine-grained identification isn't always easy, and therefore refreshing in response to every executed command often makes more sense.

Advantages of Using the Command Pattern.

Using the Command pattern has a number of advantages.

1.) One advantage is actually mentioned above. It's possible to **describe the functional part** of a program simply as a list of commands it uses.

Behind the simple program above, with a small list of commands, there can be a hundred functions that do all the work —then their implementations. Converting everything to a simple notation of a few commands makes everything easier.

The notation is so simple that project management, management, and the client can work with it. At the same time, it's essentially the program's code—complete except for the implementation (((o:

2.) The rule of this pattern is that nothing in the software can trigger activity except a command. There's an initialization, and then only commands. A single-threaded program is then **determined** by the list of commands it executes. If we invoke them again, the exact same code runs without exception. This roughly holds for multithreaded programs as well.

If we display the executed commands, we can see exactly what the program is doing. We can also write the result into the arguments—--ERROR, --INFO, etc.—and then maybe highlight it in color. If code is executed outside the program's immediate context, we can have a command to write such information. The output will appear near the command that triggered it - not like in traditional logs, where is hard to find context of execution.

3.) Functionality set. As already stated above: the application is a list of commands. Add any number of command definitions, remove any number, and you'll always have a valid application. The set might be silly, but it doesn't change the fact that the application is always functional and linked only from the commands we registered.

4.) Commands **integrate directly into the UI**; no special glue is needed. In QML, for example:

```
CadCmdButton { cmd: "cad_draw_line_pro"; visible: qmlInterface.activeFile; iconSource:
"qrc:/cad_icons/resource/icons/create_line.png"; infolineRef: infoline }
```

The button thus executes the command `cad_draw_line_pro`. See the implementation below.

```
Button {
    id: root
    property string iconSource: ""
    property string cmd: ""
    property Item infolineRef: null

    onClicked: qmlInterface.callCmd(cmd)

    contentItem: Image { anchors.fill: parent; source: root.iconSource; fillMode:
Image.PreserveAspectFit }

    MouseArea { anchors.fill: parent; acceptedButtons: Qt.RightButton
        onClicked: { if (root.infolineRef) root.infolineRef.text = root.cmd } }
}
```

Note the right-click implementation. It displays the command hidden under the button. In normal code, you'd at best have a pointer to the function to be invoked. In Qt it's better, but still just the name of a meta-function to call. And what can you do with that—you can't copy it and paste it into the command line.

5.) Documentation, help.Document the commands, and you have documentation for the entire program. Similar to the point above, this documentation can then be opened for any GUI button.

6.) Command-usage **statistics**. Some commands may never have been used. Time spent executing them is also a significant clue for command optimization.

7.) Undo. In the example above it's enough to create reverse commands. For a successful line creation, the inverse is `delete_object ID`; for deletion, the inverse is `mode_place_line; xy X Y; xy X Y`. Sometimes it's better to make a special undo command: `create_line X0 Y0 X1 Y1`.

So for a recorded command that has been executed, it's enough to generate the inverse commands and you have undo instantly.

9.) Tests. In the program, anything you do has to go through a command, so tests are basically just command calls. You can record them, generate them, whatever. The commands even check the results for you. Forget about clicking around.

8.) Refresh control. No one changes anything in the program except a command. Therefore, after executing a command is a good place for a simple refresh. True—it doesn't replace reactive object refresh, which can limit updates only to what actually changed.

9.) Script. Any program can be controlled from the outside this way, e.g., from another program. Feeding a text stream as input is a trivial matter.

The simplest program is "replay a recording," or we can modify the recording.

10.) A message received by a server can be interpreted as a command. More than 90% of servers use one huge switch. Registered commands enable **modularization**.

11.) A language model can convert any speech into a list of commands. A program that knows how to work with commands becomes instantly **voice-controllable**.

```
prompt: you have the command place_line X_BEG Y_BEG X_END Y_END<br/>
generate a set of commands: draw a smiley with a square head of size 10 centered at the origin
```

```
# REPLY:
# Head (square 10x10)
place_line -5 -5  5 -5
place_line  5 -5  5  5
place_line  5  5 -5  5
place_line -5  5 -5 -5

# Left eye
place_line -3  2 -2  2
place_line -2  2 -2  3
place_line -2  3 -3  3
place_line -3  3 -3  2

# Right eye
place_line  2  2  3  2
place_line  3  2  3  3
place_line  3  3  2  3
place_line  2  3  2  2

# Smile (approximate arc left to right)
place_line -3 -2 -2 -3
place_line -2 -3  2 -3
place_line  2 -3  3 -2
```

```
prompt, previous context: and if you can do a polyline with an unlimited number of points?
```

```
# REPLY:
# Head (square 10x10, center 0,0)
place_polyline -5 -5   5 -5   5  5   -5  5   -5 -5

# Left eye (small 1x1 square)
```

```
place_polyline -3 2   -2 2   -2 3   -3 3   -3 2

# Right eye (small 1x1 square)
place_polyline  2 2    3 2    3 3    2 3    2 2

# Smile (arc along the bottom from left to right)
place_polyline -3 -2  -2 -3  0 -3.5  2 -3  3 -2
```

Voice control of the program is a forward effect. At the time I implemented many commands, there were no language models, yet they still later served for voice control of the program.

Conclusion.

Truly the most useful pattern. You don’t need to use any other, and your application will reach a solid basic shape with a number of side effects.

In practice, it’s rarely used except in a few traditional applications: command-line operating systems, CAD software (though often being phased out there), and SQL. In all these areas, it has clear benefits—but in fact, it could serve as the core of any program.

The reasons it isn’t used more widely are:

- the botched design of the canonized pattern and the resulting informational noise,
- programmers’ distrust of what looks like a strange primitive script—though they’ll happily use SQL (o:,
- inexperience—you need to try it for a while before you can design the right set of commands.

Maybe someone will read this someday, and maybe it will be of use. But I have no illusions. It won’t change anything, even though it would make programming—and the subsequent use of software—easier.

Commentary by Chat GPT.

I understand. You’re right — it’s often not about whether the idea is correct or useful, but rather about habits, inertia, and also what once got “canonized” in textbooks and then kept spreading.

The Command pattern, as you describe it (with arguments, simple registration, naturally compatible with both UI and CLI), makes a huge amount of sense. But most people stop at the stripped-down, academic version where the practical spirit was lost.

It’s a bit like SQL — if it had only ever been shown as “classes with methods for queries,” no one would have realized the power of it being essentially a command language that describes actions.

So yes, it’s probably best not to have illusions. But even if it doesn’t shift the whole community, it can still be, for you (and a few others who discover it), a way to build cleaner, more flexible, and more controllable software.