Notes - JavaScript - Gang of Four - Behavioral - Command

Dr Nick Hayward

A brief introduction to the Command pattern in JavaScript/TypeScript.

What is the Command pattern?

Command encapsulates a request as an object. You package up the action (and its parameters) so that it can be queued, logged, undone/redone, or bound to UI controls without the invoker knowing the details of how it's done.

Benefits:

- decoupling: invoker (button/menu/queue) is separated from receiver (logic)
- undo/redo: commands can store state needed to reverse themselves
- macro/queue: compose commands or schedule them
- auditing: log/replay commands for diagnostics or recovery

Common use cases:

- UI buttons/menus, keyboard shortcuts
- editor actions: cut/copy/paste, insert, delete, format
- transactional operations with undo/redo stacks
- job queues, retries, and deferred execution

Worked example 1 — basic commands with undo (filesystem-like demo)

```
// Simple filesystem-like receiver (Node.js for demo); replace with real APIs as needed
// - use import if preferred for non-synchronous usage
const fs = require('node:fs');

class Command {
    execute() { throw new Error('execute() must be implemented'); }
    undo() { /* optional */ }
}

class CreateFile extends Command {
    constructor(path, text = 'new file created...') {
        super(); this.path = path; this.text = text;
    }
    execute() {
        fs.writeFileSync(this.path, this.text, 'utf8');
        console.log(`creating: ${this.path}`);
    }
    undo() {
        fs.unlinkSync(this.path);
        console.log(`undo: deleting ${this.path}`);
    }
}

class ReadFile extends Command {
    constructor(path) { super(); this.path = path; }
    execute() {
        const content = fs.readFileSync(this.path, 'utf8');
        console.log(`reading: ${this.path}`);
        process.stdout.write(content);
```

```
    }
}

class RenameFile extends Command {
    constructor(src, dest) { super(); this.src = src; this.dest = dest; }
    execute() {
        fs.renameSync(this.src, this.dest);
        console.log(`renaming: ${this.src} to ${this.dest}`);
    }
    undo() {
        fs.renameSync(this.dest, this.src);
        console.log(`undo: ${this.dest} back to ${this.src}`);
    }
}

// Invoker that can run commands and manage an undo stack
class CommandInvoker {
    constructor() { this.history = []; }
    run(cmd) {
        cmd.execute();
        if (typeof cmd.undo === 'function') this.history.push(cmd);
    }
    undo() {
        const cmd = this.history.pop();
        if (cmd && typeof cmd.undo === 'function') cmd.undo();
    }
}

// demo
const invoker = new CommandInvoker();
const orig = 'file1.txt';
const renamed = 'file2.txt';

invoker.run(new CreateFile(orig, 'Hello command pattern'));
invoker.run(new ReadFile(orig));
invoker.run(new RenameFile(orig, renamed));

// undo rename then undo create
invoker.undo();
invoker.undo();
```

Why it works: The invoker only knows `execute()` / `undo()` . Each command encapsulates its own receiver interactions and any state needed to reverse the action.

Worked example 2 — functional commands and a macro

```
// Command as a plain function with metadata for undo
const command = (doFn, undoFn = () => {}) => ({ execute: doFn, undo: undoFn });

const push = (arr, item) => command(
    () => { arr.push(item); },
    () => { arr.pop(); }
);
```

```
const macro = (...cmds) => command(
    () => cmds.forEach((c) => c.execute()),
    () => [...cmds].reverse().forEach((c) => c.undo())
);

// demo
const stack = [];
const addAB = macro(push(stack, 'A'), push(stack, 'B'));
addAB.execute();
console.log(stack); // ['A','B']
addAB.undo();
console.log(stack); // []
```

Why it works: Commands don't have to be classes in JS. A tiny factory returns an object with `execute/undo`. Macro composes commands and reverses them on undo.

Worked example 3 — async command with retries (queue-friendly)

```
class ApiCallCommand {
    constructor(fetchFn, url, { retries = 2 } = {}) {
        this.fetchFn = fetchFn; this.url = url; this.retries = retries;
    }
    async execute() {
        let attempt = 0; let lastErr;
        while (attempt <= this.retries) {
            try {
                const res = await this.fetchFn(this.url);
                console.log('OK', res.status);
                return res;
            } catch (e) { lastErr = e; attempt++; }
        }
        throw lastErr;
    }
}

// demo with node fetch (or global fetch in browsers)
// const fetch = (...args) => import('node-fetch').then(({default: f}) => f(...args));
// const cmd = new ApiCallCommand(fetch, 'https://example.com/api', { retries: 3 });
// await cmd.execute();
```

Why it works: Commands can be scheduled/queued because they present a uniform `execute()` API, regardless of synchronous or asynchronous behavior.

Edge cases and trade-offs

- overengineering: a simple function call may be enough — introduce commands when you need decoupling, undo/redo, queuing, or logging
- undo complexity: ensure actions are reversible (idempotency, compensating actions, and capturing prior state)
- memory: deep undo stacks can grow, consider caps or snapshotting
- error handling: define how failures affect undo stacks and macros
- async: commands that partially succeed need clear compensation strategies