Command Design Pattern

Ignat Alex-Matei
Ignat Dragoș-Mihai
Măierean Mircea

Table of contents

🧠 *Imagine you're designing a smart home system.*

You have a central controller system – like a voice assistant or an app – and a bunch of smart devices: lights, fans, blinds, coffee machines. Your goal is simple: press a button or say a command, and the device does what you want.

At first, it's easy. You write a method for each device:

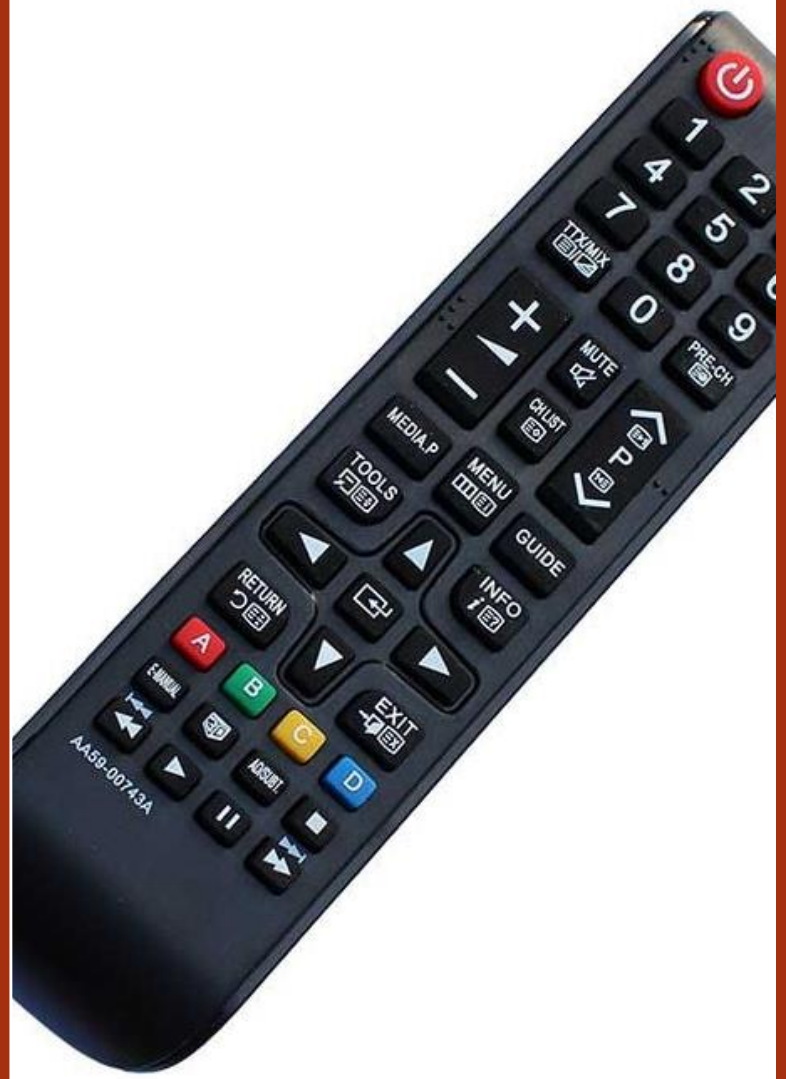
- *turnOnLight()*
- *startCoffeeMachine()*
- *openBlinds()*

But then things get *complicated*…

Your users want **to schedule commands**: ☕ "Start the coffee machine at 7 AM."

They want to **undo actions**: 💡 "Oops, I didn't mean to turn on that light!"

They want to **group actions together**: 🏠 "I'm leaving – turn everything off!"



What is Command?

- Command is a behavioural design pattern that turns a request into a stand-alone object called a command.
- With the help of this pattern, each component of a request can be captured individually, including the object that owns the method, the parameters for the method, and the method itself.
- This design pattern has lots of applications, such as passing, queuing or logging requests, or support operations like undo-redo.

Structure

1. Command Interface

- Declares a common method, *execute*().
- Sets the standard for all commands.

2. Concrete Command Classes

- Implement *Command* Interface.
- Encapsulate a request as an object that can be executed later.

3. Caller (Invoker)

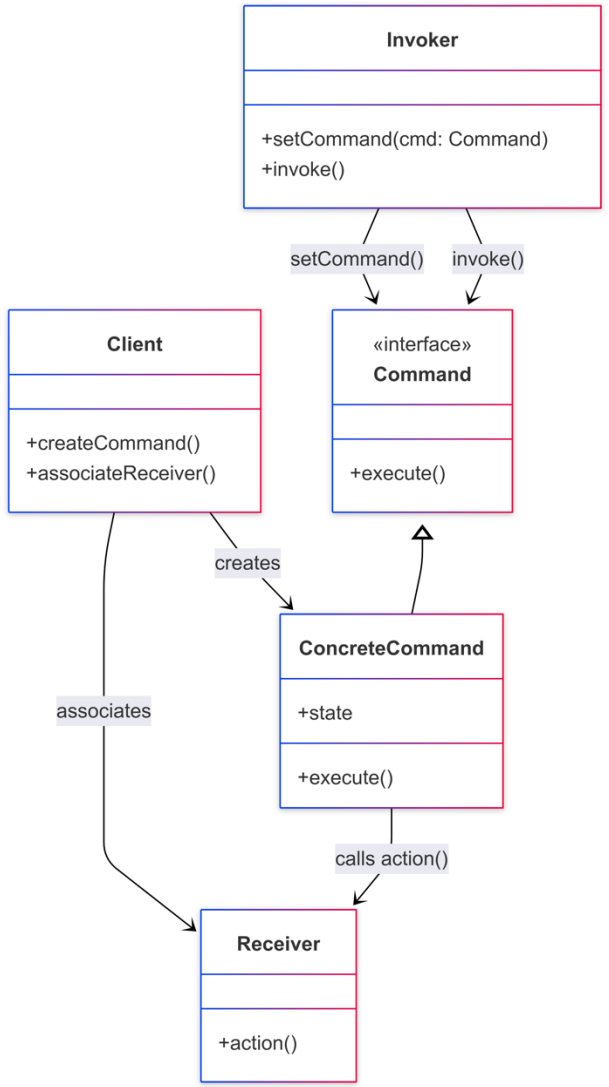
- Is the one responsible for initiating command execution.
- Holds a command without specifics of how each command works.

4. Receiver

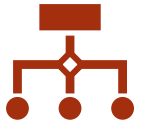
- Knows how to perform the actual operation associated with a command.
- Understand the specific tasks mentioned in commands.
- Separates responsibilities.

5. Client

- Creates the concrete command objects.
- Associates them with a receiver.

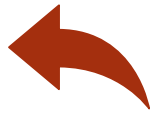


Usages for Command



Parameterize Objects with Operations

Use the Command pattern to **parameterize methods with different requests**, queues, or operations to be executed at different times.

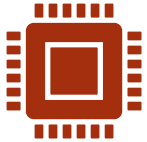


Undo/Redo Functionality

Perfect for implementing **undo/redo mechanisms**. Each command object stores the details required to **reverse or reapply** an action.

Queueing or Logging Operations

Useful for **queuing requests, scheduling tasks**, or **logging user actions** to be executed later.



Decoupling Sender and Receiver

Applied for **decoupling the object invoking an operation from the one that performs it**, making code more flexible and extensible.

Macros / Composite Commands

Great for **composing multiple commands into a single action** (e.g., macro recording in apps).

Pros and Cons

PROS

01 Promotes separation of concerns

02 Encapsulates operations as objects

03 Facilitates extensibility and flexibility

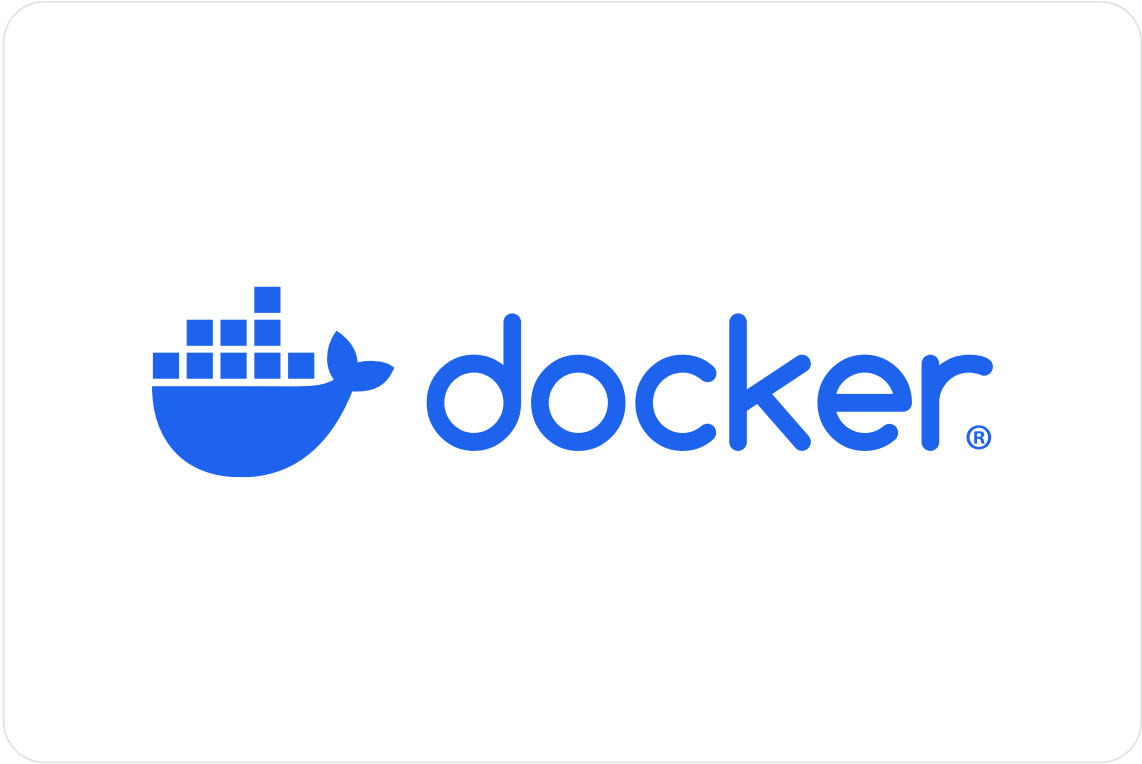
CONS

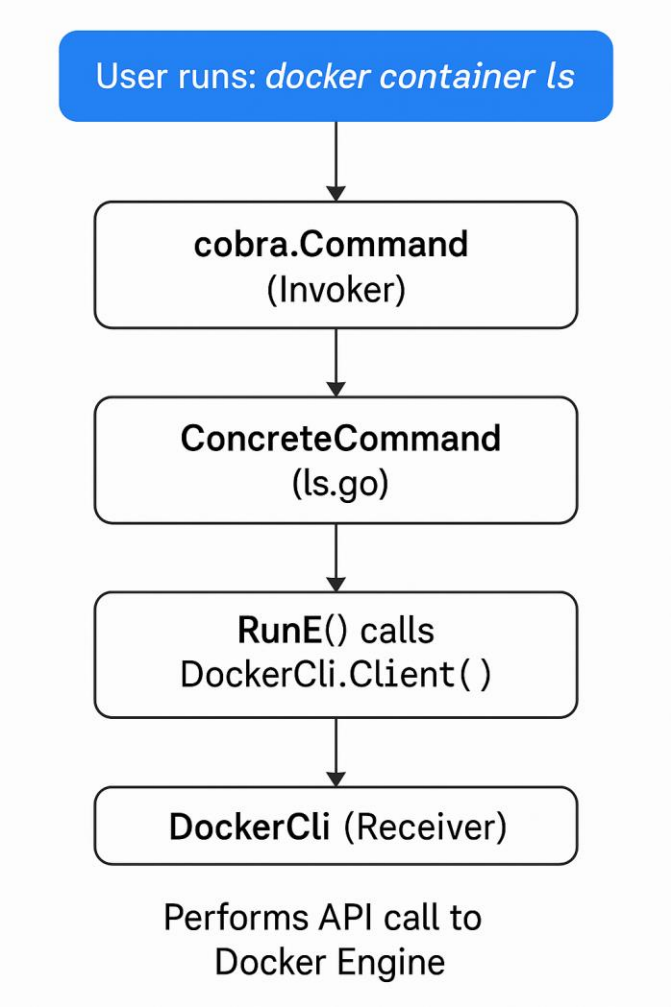
May introduce unnecessary abstractions 01

Can increase the number of classes 02

Management of command lifecycle adds complexity 03

Example 1 - How is the command pattern implemented in the Docker CLI




User runs: *docker container ls*
cobra.Command
(Invoker)
ConcreteCommand
(ls.go)
RunE() calls
DockerCli.Client()
DockerCli (Receiver)
Performs API call to
Docker Engine

The Docker CLI

- Docker CLI uses the Cobra framework in Go to implement the Command pattern. This pattern decouples the command interface from its implementation, allowing Docker to create a consistent user experience while maintaining separation of concerns.
- The Command pattern enables Docker to easily extend its functionality through plugins, handle complex command composition (as seen in docker-compose), and maintain a clean separation between the user interface and the underlying execution logic.

The Command and Concrete Command

- Docker CLI uses Cobra for the command interface, the *RunE* field is used for the executing a method.
- Concrete commands are implemented in separate files inside subfolders that are specific for each concern.

```go
// NewPsCommand creates a new cobra.Command for `docker ps`
func NewPsCommand(dockerCLI command.Cli) *cobra.Command {
	options := psOptions{filter: opts.NewFilterOpt()}

	cmd := &cobra.Command{
		Use:   "ps [OPTIONS]",
		Short: "List containers",
		Args:  cli.NoArgs,
		RunE: func(cmd *cobra.Command, args []string) error {
			options.sizeChanged = cmd.Flags().Changed("size")
			return runPs(cmd.Context(), dockerCLI, &options)
		},
		Annotations: map[string]string{
			"category-top": "3",
			"aliases":      "docker container ls, docker container list, docker container ps, docker ps",
		},
		ValidArgsFunction: completion.NoComplete,
	}

	flags := cmd.Flags()

	flags.BoolVarP(&options.quiet, "quiet", "q", false, "Only display container IDs")
	flags.BoolVarP(&options.size, "size", "s", false, "Display total file sizes")
	flags.BoolVarP(&options.all, "all", "a", false, "Show all containers (default shows just running)")
	flags.BoolVar(&options.noTrunc, "no-trunc", false, "Don't truncate output")
	flags.BoolVarP(&options.nLatest, "latest", "l", false, "Show the latest created container (includes all states)")
	flags.IntVarP(&options.last, "last", "n", -1, "Show n last created containers (includes all states)")
	flags.StringVar(&options.format, "format", "", flagsHelper.FormatHelp)
	flags.VarP(&options.filter, "filter", "f", "Filter output based on conditions provided")

	return cmd
}
```

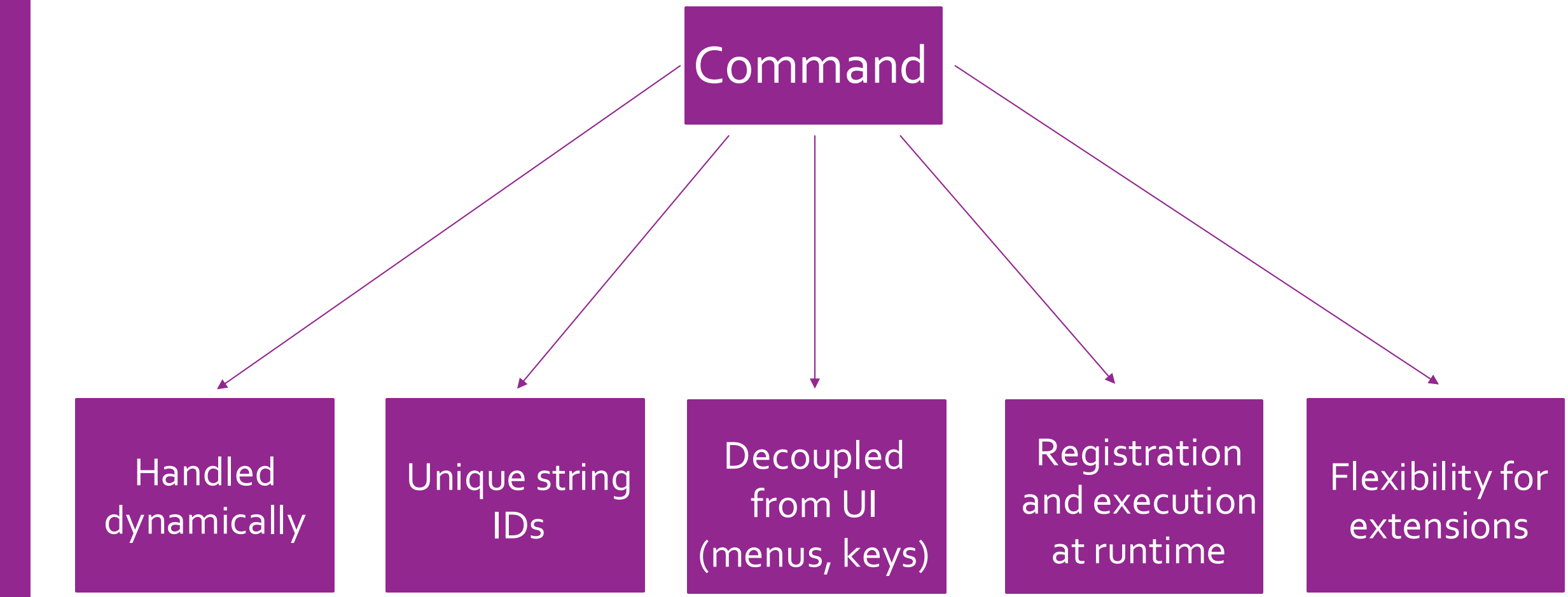
The receiver and invoker

- In here the receiver that does the actual business logic is the DockerCli, which is an instance of the docker client, it has access to different parts of the app.
- The invoker here is represented by the *execute()* method inside the cobra command.

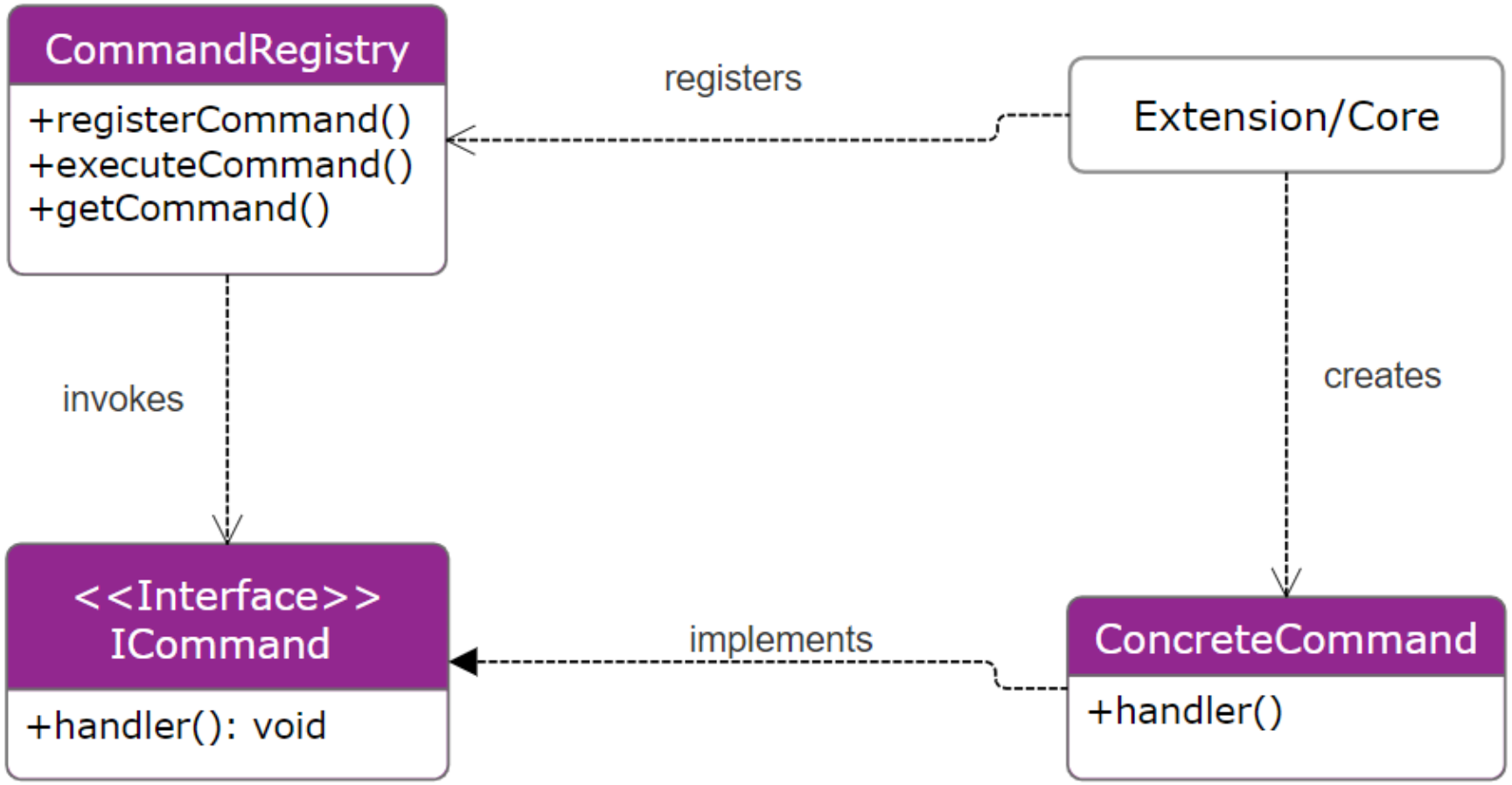
Benefits

- **Extensibility**
  - New commands can be added easily without modifying existing logic.
  - Plugins and features can be modularly developed.
- **Separation of Concerns**
  - Each command focuses only on what it needs to do.
  - Business logic is handled by the DockerCli (Receiver), not the command itself.
- **Maintainability**
  - Each command lives in its own file/folder, making it easy to debug and modify independently.
  - The CLI stays organized and scalable as more functionality is added.
- **Testability**
  - Commands can be tested in isolation using mock DockerCli receivers.
  - Helps with unit testing without spinning up a full Docker daemon.

Example – VS Code Command Registry



Command Design Pattern in VS Code Commands



How do VS Code Commands work?

```
interface CommandHandler {
    callback: Function;
    thisArg: any;
    metadata?: ICommandMetadata;
    extension?: IExtensionDescription;
}

export interface ArgumentProcessor {
    processArgument(arg: any, extension: IExtensionDescription | undefined): any;
}
```

CommandHandler: the **interface** for the **command**
ArgumentProcessor: helper that processes the arguments of the command

How do VS Code Commands work?

```
export class ExtHostCommands implements ExtHostCommandsShape {

    readonly _serviceBrand: undefined;
    #proxy: MainThreadCommandsShape;
    private readonly _commands = new Map<string, CommandHandler>();
    #telemetry: MainThreadTelemetryShape;
    private readonly _logService: ILogService;
    readonly #extHostTelemetry: IExtHostTelemetry;
    private readonly _argumentProcessors: ArgumentProcessor[];
    readonly converter: CommandsConverter;

    registerCommand(global: boolean,
                    id: string, callback: <T>(...args: any[]) => T | Thenable<T>,
                    thisArg?: any,
                    metadata?: ICommandMetadata,
                    extension?: IExtensionDescription): extHostTypes.Disposable {}
    executeCommand<T>(id: string, ...args: any[]): Promise<T> {}
    private async _doExecuteCommand<T>(id: string, args: any[], retry: boolean): Promise<T> {}
    getCommands(filterUnderscoreCommands: boolean = false): Promise<string[]> {}
}
```

ExtHostCommands: the **invoker** which calls and handles the commands, mapping a **unique ID** to each one

How do VS Code Commands work?

```
export class CommandsConverter implements extHostTypeConverter.Command.ICommandsConverter {

    readonly delegatingCommandId: string = `__vsc${generateUuid()}`;
    private readonly _cache = new Map<string, vscode.Command>();
    private _cachIdPool = 0;

    toInternal(command: vscode.Command | undefined, disposables: DisposableStore): ICommandDto | undefined {}
    fromInternal(command: ICommandDto): vscode.Command | undefined {}

    getActualCommand(...args: any[]): vscode.Command | undefined {}
    private _executeConvertedCommand<R>(...args: any[]): Promise<R> {}
}
```

CommandsConverter: helps with **internal** and **API** commands, by converting from one another

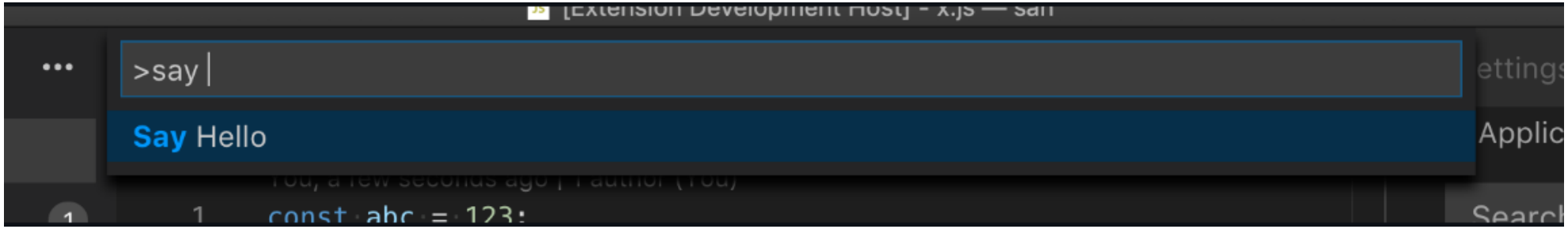
Implementing a concrete command

```
import * as vscode from 'vscode';

export function activate(context: vscode.ExtensionContext) {
  const command = 'myExtension.sayHello';

  const commandHandler = (name: string = 'world') => {
    console.log(`Hello ${name}!!!`);
  };

  context.subscriptions.push(vscode.commands.registerCommand(command, commandHandler));
}
```



Good use of the Command Design Pattern?

Decoupling from UI	Flexibility & Extensibility	Centralized Management	Dynamic Behavior	Improved Maintainability
• Commands independent from UI elements (menus, shortcuts) • Easy to change UI without modifying command logic	• Extensions can easily add or modify commands • Commands can be registered or unregistered dynamically	• Simplified command handling, logging, monitorizing • Easier debugging and maintenance	• Commands invoked dynamically at runtime • Facilitates undo/redo, macros, and batch executions	• Clear separation of concerns • Reduced coupling makes adding features an easy task

References:

Theory:
https://en.wikipedia.org/wiki/Command_pattern
https://medium.com/swlh/command-pattern-what-it-is-and-how-to-use-it-7ccbc810266d
https://www.tutorialspoint.com/design_pattern/command_pattern.htm
https://www.geeksforgeeks.org/command-pattern/

Example 1:
https://github.com/docker/cli/blob/master/cli/command/cli.go
https://github.com/docker/cli/blob/master/cli/command/container/list.go
https://github.com/spf13/cobra

Example 2:
https://code.visualstudio.com/api/extension-guides/command
https://github.com/microsoft/vscode/blob/main/src/vs/workbench/api/common/extHostCommands.ts
https://code.visualstudio.com/api/references/contribution-points#contributes.commands