### Command Design Pattern [Layman’s View]

The Command Design Pattern is a behavioral design pattern that turns a request into a stand-alone object containing all information about the request. This transformation allows you to parameterize methods with different requests, delay or queue a request's execution, and support undoable operations. Let's delve into this pattern in a detailed manner, starting with a layman's understanding before moving onto its technical aspects, including when to use it, when not to use it, pitfalls, and a code example.

Imagine you're at a restaurant. You decide what you want to order and convey your order to the waiter. The waiter takes your request and delivers it to the kitchen where your order is prepared. In this scenario, your order is a command. It contains all the necessary information (what you want to eat or drink) and is handled by different objects at different times (waiter and chef).

In software terms, the Command pattern encapsulates a request as an object, thereby letting you parameterize clients with queues, requests, and operations. ***It provides the means to separate the responsibility of issuing commands from the responsibility of executing commands***, allowing for a wide range of flexibility and control in the execution of these commands.

### Code Example

public interface Command {

void execute();

}

// Concrete Command to open a file

public class OpenFileCommand implements Command {

private FileSystemReceiver fileSystem;

public OpenFileCommand(FileSystemReceiver fs){

this.fileSystem = fs;

}

@Override

public void execute(){

// open command is forwarding request to openFile method

this.fileSystem.openFile();

}

}

// Similarly, you can create SaveFileCommand and CloseFileCommand

We need a receiver, the FileSystemReceiver interface, and its implementations:

public interface FileSystemReceiver {

void openFile();

void writeFile();

void closeFile();

}

// UnixFileSystemReceiver and WindowsFileSystemReceiver implement FileSystemReceiver

Invoker class that invokes the command:

public class FileInvoker {

    private Command command;

    public FileInvoker(Command c){

        this.command = c;

    }

    public void execute(){

        this.command.execute();

    }

}

Client code:

public class Client {

    public static void main(String[] args) {

        FileSystemReceiver fs = FileSystemReceiverUtil.getUnderlyingFileSystem();

        // Creating the command and associating it with the receiver

        Command openFileCommand = new OpenFileCommand(fs);

        // Creating invoker and associate it with the command

        FileInvoker file = new FileInvoker(openFileCommand);

        // Perform action on the invoker object

        file.execute();

    }

}

### When to Use the Command Design Pattern

* **Decoupling Invokers and Receivers:** Use it when you need to decouple the object that invokes the operation from the one that knows how to perform it.
* **Parameterization of Objects:** When you need to specify, queue, and execute requests at different times.
* **Support for Undo Operations:** It's useful when you need to support undo-able operations.
* **Extending Systems:** When you need to extend systems by adding new commands, without changing existing code.

### When Not to Use the Command Design Pattern

* **Simple Commands:** For simple commands or requests, using this pattern might be an overkill, leading to unnecessary complexity.
* **Performance Concerns:** If performance is a critical factor, the additional layer of abstraction and the objects it introduces can impact performance.

**Pitfalls**

* **Overhead:** Introducing too many command classes can lead to an increase in complexity and maintenance overhead.
* **Underutilization:** Implementing the Command pattern without sufficient reason or for very simple operations can complicate the architecture unnecessarily.
* **Memory Footprint:** If not managed properly, especially with a large number of commands or a long history of commands for undo functionality, it can lead to a significant memory footprint.