Assignment 3 | Command Pattern

Introduction

The Command Pattern is a behavioral design pattern that turns a request into a stand-alone object containing all information about the request.
The Command Pattern encapsulates a request as an object, allowing for the separation of sender and receiver. It enables you to parameterize objects with different requests, queue or log requests, and support undoable operations.
This pattern promotes "invocation of a method on an object" to full object status, essentially creating an object-oriented callback mechanism. By doing so, it decouples the object that invokes the operation from the one that knows how to perform it, providing flexibility and extensibility in software design.

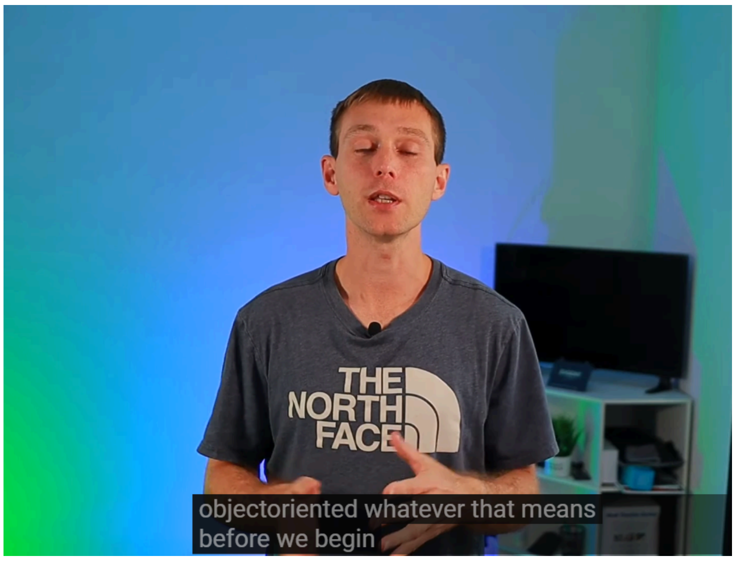
Youtube Videos that best describe the Command Pattern

It explains how the Command pattern can help manage delayed execution, undo/redo systems, and sequences of actions. The video uses a cooking robot analogy and a game development example to illustrate the pattern's implementation and benefits.

(click photo for link)

Assignment 3 | Command Pattern

This video explains the Command Design Pattern using a remote control programming example. It demonstrates how the pattern helps reduce if-else statements and makes code more object-oriented. The video covers the implementation of the Command interface, creating command classes, and setting up a remote control to execute commands.



(click photo for link)

Rationale

Decoupling: The primary rationale for using the Command Pattern is to decouple the object that invokes an operation (the invoker) from the object that performs the operation (the receiver). This separation provides flexibility and extensibility in software design.

Encapsulation of Requests: The pattern encapsulates a request as a standalone object, containing all information about the request. This allows for:

- Parameterization of objects: Different commands can be created with different parameters without changing the invoker[1].
- Queueing of requests: Commands can be stored and executed later.
- Logging of operations: Each command execution can be easily logged.
- Undo/Redo functionality: Commands can store state for undoing operations.

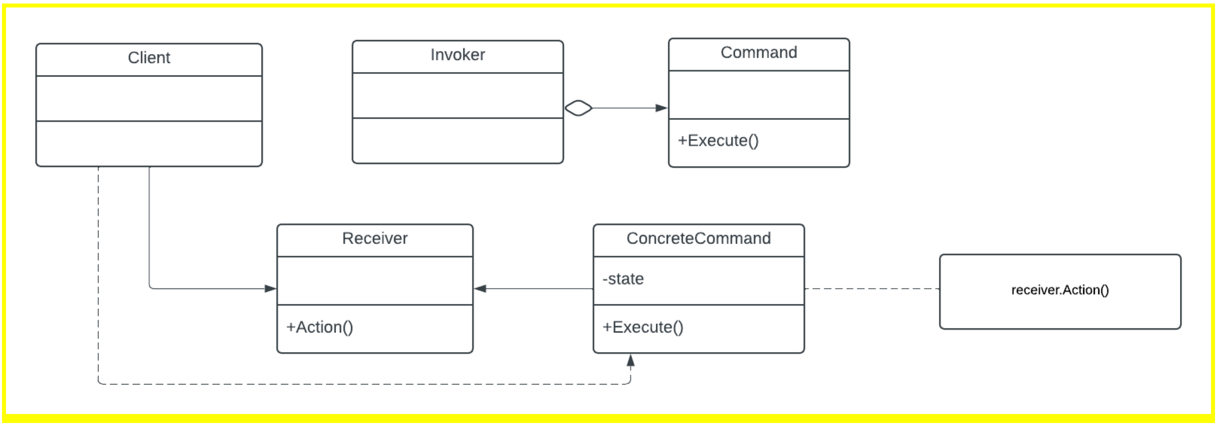
Flexibility in Execution: By turning requests into command objects, the pattern enables:

- Delayed execution: Commands can be stored and executed at a later time.
- Remote execution: Commands can be sent to other parts of a system for execution.
- Composite commands: Multiple commands can be combined into macro commands.

Extensibility: New commands can be added without modifying existing code, adhering to the Open/Closed Principle. This makes it easier to extend and maintain the system.

Assignment 3 | Command Pattern

UML Diagram



Client: The client is responsible for creating the ConcreteCommand and associating it with the Receiver. It then assigns the command to the Invoker for execution.

Invoker: The invoker keeps a reference to the command and triggers its execution. It doesn't know the specifics of what the command does, only that it can call Execute() on it.

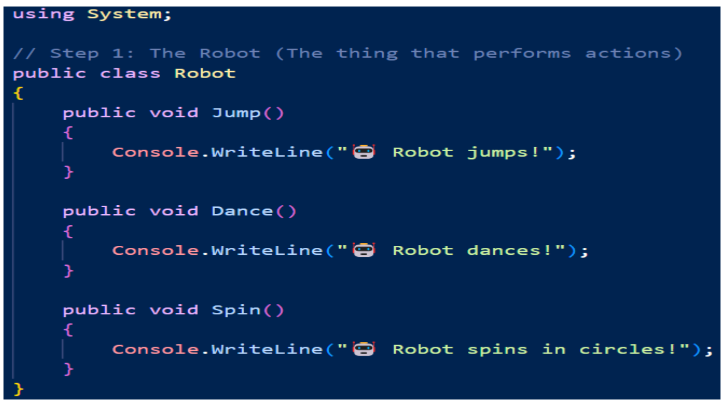
Command (Interface or Abstract Class): This is an interface or abstract class that declares an Execute() method, which all concrete commands must implement.

ConcreteCommand: This class implements the Command interface and is tied to a Receiver object. It calls one or more actions on the receiver when its Execute() method is invoked.

Receiver: The receiver is the object that performs the actual work of the command. It contains the business logic of the application.

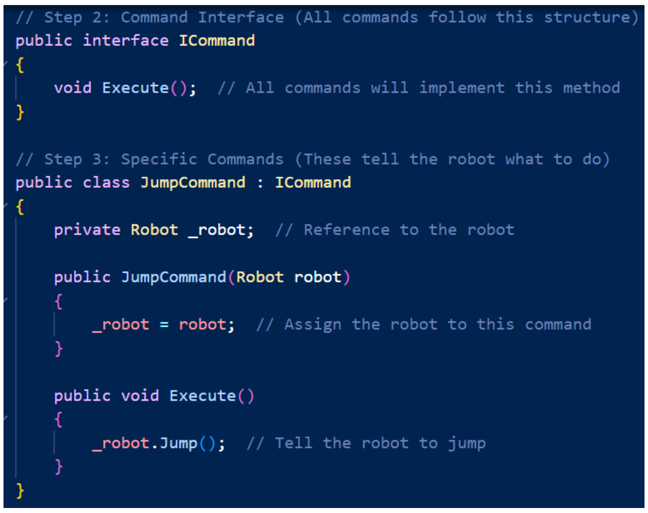
Code Example

Simple Sample Code (Sample ← click here)

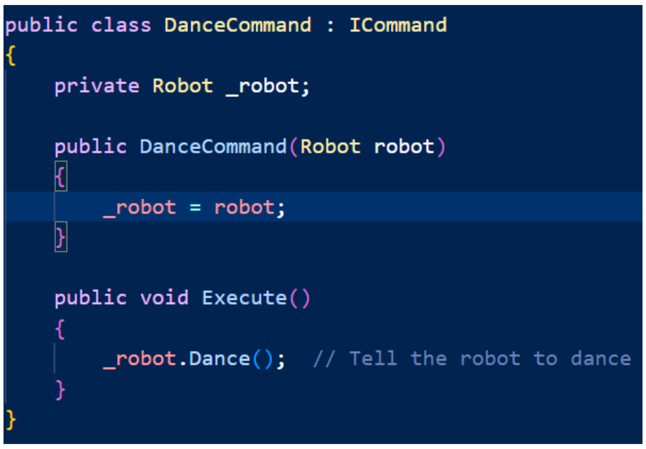


```
using System;

// Step 1: The Robot (The thing that performs actions)
public class Robot
{
    public void Jump()
    {
        Console.WriteLine("🤖 Robot jumps!");
    }

    public void Dance()
    {
        Console.WriteLine("🤖 Robot dances!");
    }

    public void Spin()
    {
        Console.WriteLine("🤖 Robot spins in circles!");
    }
}
```

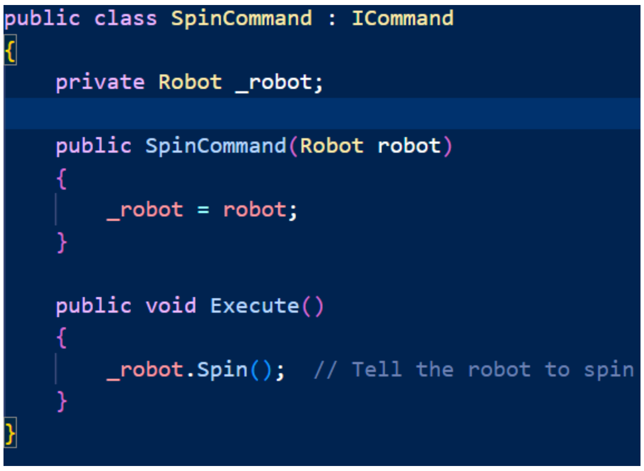
Assignment 3 | Command Pattern



```
// Step 2: Command Interface (All commands follow this structure)
public interface ICommand
{
    void Execute();  // All commands will implement this method
}

// Step 3: Specific Commands (These tell the robot what to do)
public class JumpCommand : ICommand
{
    private Robot _robot;  // Reference to the robot

    public JumpCommand(Robot robot)
    {
        _robot = robot;  // Assign the robot to this command
    }

    public void Execute()
    {
        _robot.Jump();  // Tell the robot to jump
    }
}
```

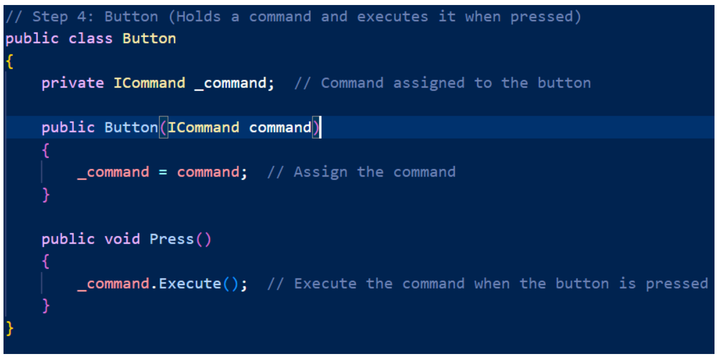


```
public class DanceCommand : ICommand
{
    private Robot _robot;

    public DanceCommand(Robot robot)
    {
        _robot = robot;
    }

    public void Execute()
    {
        _robot.Dance();  // Tell the robot to dance
    }
}
```

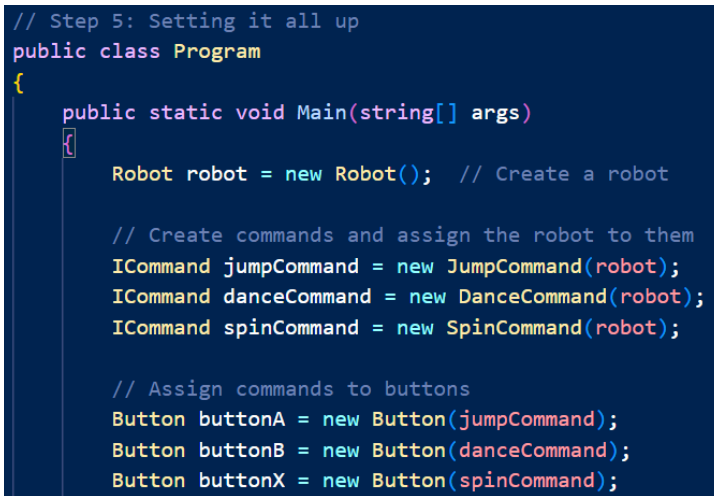


```
public class SpinCommand : ICommand
{
    private Robot _robot;

    public SpinCommand(Robot robot)
    {
        _robot = robot;
    }

    public void Execute()
    {
        _robot.Spin();  // Tell the robot to spin
    }
}
```

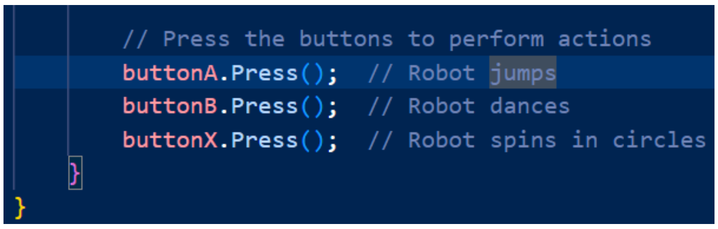
Assignment 3 | Command Pattern



```
// Step 4: Button (Holds a command and executes it when pressed)
public class Button
{
    private ICommand _command;  // Command assigned to the button

    public Button(ICommand command)
    {
        _command = command;  // Assign the command
    }

    public void Press()
    {
        _command.Execute();  // Execute the command when the button is pressed
    }
}
```



```
// Step 5: Setting it all up
public class Program
{
    public static void Main(string[] args)
    {
        Robot robot = new Robot();  // Create a robot

        // Create commands and assign the robot to them
        ICommand jumpCommand = new JumpCommand(robot);
        ICommand danceCommand = new DanceCommand(robot);
        ICommand spinCommand = new SpinCommand(robot);

        // Assign commands to buttons
        Button buttonA = new Button(jumpCommand);
        Button buttonB = new Button(danceCommand);
        Button buttonX = new Button(spinCommand);
```



```
        // Press the buttons to perform actions
        buttonA.Press();  // Robot jumps
        buttonB.Press();  // Robot dances
        buttonX.Press();  // Robot spins in circles
    }
}
```

Common Usage in the Industry

The Command Pattern is widely used in various industries and applications.

1. GUI Development

In graphical user interface (GUI) frameworks, the Command Pattern is extensively used to handle user actions:

- Buttons and Menu Items: Each button or menu item is associated with a specific command object. When clicked, the command's execute() method is called, performing the desired action.
- Undo/Redo Functionality: Commands can store state for undoing operations, allowing for easy implementation of undo and redo features in applications like text editors or graphic design software.

Assignment 3 | Command Pattern

2. Transaction Management in Financial Systems

Financial systems often use the Command Pattern to manage complex transactions:

- Atomic Operations: Each financial transaction (e.g., transfer, withdrawal) is encapsulated as a command object, ensuring atomicity and consistency.
- Transaction Logging: Commands can be logged before execution, providing an audit trail and enabling system recovery in case of failures.
- Scheduled Transactions: Commands can be queued for later execution, allowing for scheduled or batch processing of financial operations.

3. Game Development

The Command Pattern is frequently employed in game development for various purposes:

- Input Handling: Player inputs are translated into command objects, allowing for easy remapping of controls and implementation of macros.
- AI Behavior: AI actions can be represented as commands, facilitating the creation of complex AI behaviors and decision-making processes.
- Replay Systems: By recording and storing sequences of commands, games can implement replay functionality, allowing players to review past gameplay sessions.

These examples demonstrate the versatility of the Command Pattern across different domains.

Complex Code Example

C# implementation demonstrating the Command Pattern for the smart home automation system:

Complex Sample Code (Sample Code ← click here)

Problem Statement

Developing a smart home automation system. The system needs to control various devices (lights, thermostat, security system) through different interfaces (mobile app, voice commands, web interface). Additionally, the system should support scheduling of commands, logging of all operations, and the ability to undo recent actions.
The challenge is to create a flexible system that can easily accommodate new devices and interfaces without major code changes, while also supporting advanced features like scheduling and undo functionality.

How the Command Pattern Solves the Problem

The Command Pattern addresses this problem by:

Assignment 3 | Command Pattern

1. Encapsulating each action as a command object, allowing for easy addition of new devices and actions.
2. Decoupling the interfaces (invokers) from the devices (receivers), enabling the addition of new interfaces without affecting existing code.
3. Facilitating the implementation of scheduling by allowing commands to be stored and executed later.
4. Enabling easy logging of all operations by logging command executions.
5. Supporting undo functionality by storing the state in commands and implementing an undo method.

Command Interface

The Command interface defines a common method, typically called Execute(), that all concrete command classes must implement. This interface establishes a standard contract for executing commands.

Concrete Command Classes

These classes implement the Command interface and encapsulate specific actions or requests. Each Concrete Command:

- Implements the Execute() method with the specific logic for that command
- Often holds a reference to the Receiver object
- May store command arguments or state for undoable operations

Receiver

The Receiver is the object that actually performs the operations associated with a command. It contains the business logic to carry out the request. When a command's Execute() method is called, it typically invokes one or more methods on the Receiver.

Invoker

The Invoker is responsible for initiating requests. It holds a command object and triggers its execution by calling the Execute() method. The Invoker:

- Doesn't know the specifics of how the command is implemented
- Can store and manage multiple commands
- May support operations like undo/redo or command history

Client

The Client creates and configures concrete Command objects. It:

- Instantiates specific Command objects
- Sets the Receiver for the command
- Assigns commands to Invokers

Assignment 3 | Command Pattern

By separating these components, the Command Pattern achieves several benefits:

1. Decoupling the sender (Invoker) from the object that performs the operation (Receiver)
2. Allowing commands to be treated as objects, which can be stored, passed around, and manipulated
3. Enabling easy extension of the system with new commands without modifying existing code
4. Supporting advanced features like undo/redo, command queuing, and logging of operations

Assignment 3 | Command Pattern

References:

SourceMaking. (n.d.). Command. *SourceMaking.* https://sourcemaking.com/design_patterns/command

Command Pattern. (n.d.). In Wikipedia. Retrieved October 19, 2024, from https://en.wikipedia.org/wiki/Command_Pattern

GeeksforGeeks. (n.d.). Command Pattern. GeeksforGeeks. https://www.geeksforgeeks.org/command-pattern/

DigitalOcean. (2022, August 4). Command Design Pattern. DigitalOcean. https://www.digitalocean.com/community/tutorials/command-design-pattern