# Command Design Pattern

Casey Scarborough

October 3, 2013

## Contents

## What is the Command Design Pattern?

Definition from Wikipedia:

> *The command pattern is a behavioral design pattern in which an object is used to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method and values for the method parameters.*

In simpler terms, the Command Design pattern gives you a way to execute commands, keep track of them, redo or undo them, along with some other features.

## Parts of the Pattern

The Command Design pattern has five main parts: a *Command* interface, a *ConcreteCommand*, a *Receiver*, an *Invoker*, and a *Client*. An explanation of these parts is as follows:

- The *Command* interface declares the methods that will be used for executing an operation.
- The *ConcreteCommand* will implement the *Command* interface and also defines a binding between a *Receiver* and an action.
- The *Client* handles the creation of the command object and will set its receiver.
- The *Invoker* asks the command to carry out the request. It actually *invokes* the command.
- The *Receiver* knows how to perform the operations associated with each request.

This may seem a little complex, but it will all come together soon enough. The following figure shows the UML diagram for the pattern.

### UML Diagram

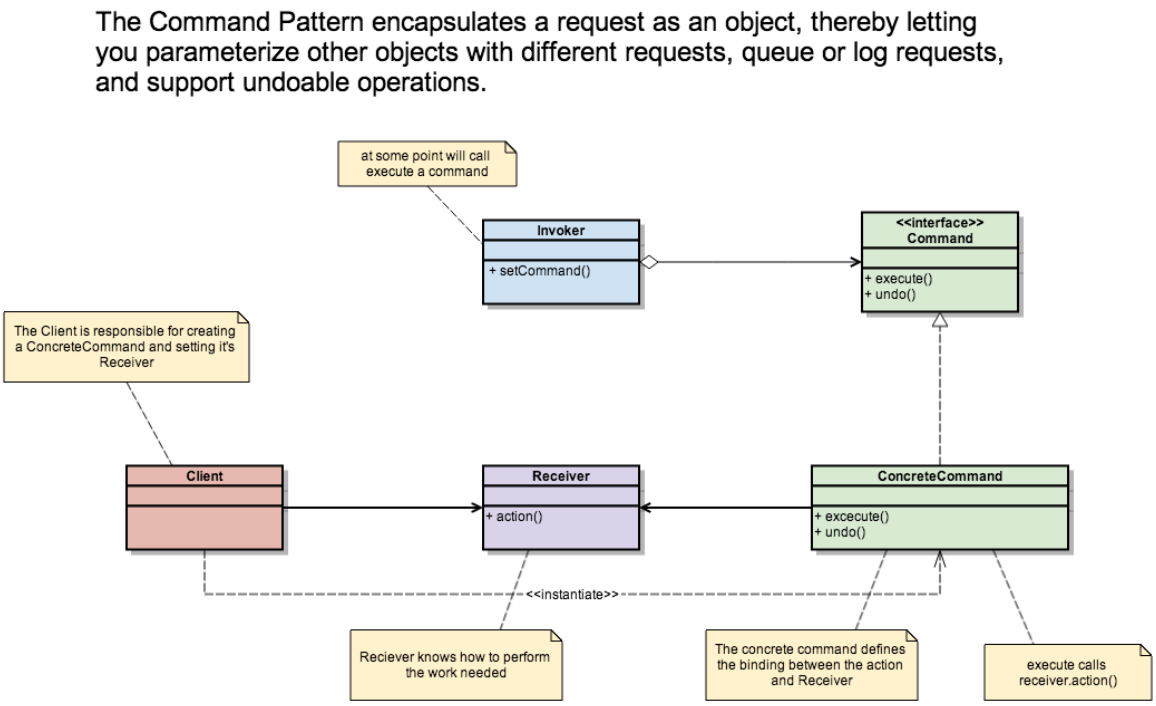


Figure 1: UML Diagram for the Command Pattern

## Pros and Cons

The Command Design pattern is useful for encapsulating requests as objects, allowing you to parameterize requests, log them, or queue them, as well as support undoable operations. The disadvantage to using this pattern is that you must create multiple small classes to store lists of commands.

## Examples

### Structural Example (Java)

CommandPattern.java

```
/**
 * This is essentially the Client. It is the main application and
 * handles the creation of the Command object and setting its receiver,
 * and then passing that command object to the invoker. The commands
 * can then be executed via the invoker.
 */
public class CommandPattern {
  public static void main(String[] args) {

    // Create a new receiver.
    Receiver receiver = new Receiver();

    // Create a new command and bind it to our receiver.
    Command command = new ConcreteCommand(receiver);

    // Create an invoker to execute commands.
    Invoker invoker = new Invoker();

    // Bind the command to our invoker and execute it.
    invoker.setCommand(command);
    invoker.executeCommand();
  }
}

/**
 * The receiver is the object which the action is being
 * performed on. It knows how to perform the operations
 * associated with carrying out the request.
 */
class Receiver {
  public void action() {
    System.out.println("Receiver.action() method has been called.");
  }
}

/**
 * This is the command interface. It sets the rules
 * that each command will have to follow. In this case,
 * every command must have an execute method.
 */
interface Command {
  public void execute();
}
```

```
/**
 * This is an actual command. It implements the Command interface
 * and it defines the binding between the receiver object and its
 * action. It implements the execute method by calling the corresponding
 * action on the receiver.
 */
class ConcreteCommand implements Command {
  private Receiver receiver;

  public ConcreteCommand(Receiver receiver) {
    this.receiver = receiver;
  }

  public void execute() {
    receiver.action();
  }
}

/**
 * The invoker asks the command to carry out the request.
 * It is bound to one specific command at any given time.
 */
class Invoker {
  private Command command;

  public void setCommand(Command command) {
    this.command = command;
  }

  public void executeCommand() {
    this.command.execute();
  }
}
```

### Structural Example (C#)

CommandPattern.cs

```
using System;

namespace CommandPatternStructural
{
    // <summary>
    // This is essentially the Client. It is the main application and
    // handles the creation of the Command object and setting its receiver,
    // and then passing that command object to the invoker. The commands
    // can then be executed via the invoker.
    // </summary>
    class CommandPattern
    {
        static void Main(string[] args)
        {
            // Create a new receiver.
            Receiver receiver = new Receiver();

            // Create a new command and bind it to our receiver.
            Command command = new ConcreteCommand(receiver);

            // Create a new invoker to execute commands.
            Invoker invoker = new Invoker();

            // Set the invoker's command and execute it.
            invoker.SetCommand(command);
            invoker.ExecuteCommand();

            // Wait for user.
            Console.ReadKey();
        }
    }

    // <summary>
    // The receiver is the object which the action is being
    // performed on. It knows how to perform the operations
    // associated with carrying out the request.
    // </summary>
    class Receiver
    {
        public void Action()
        {
            Console.Write("Receiver.Action() has been called!");
        }
    }
```

```
    // <summary>
    // The receiver is the object which the action is being
    // performed on. It knows how to perform the operations
    // associated with carrying out the request.
    // </summary>
    public interface Command
    {
        void Execute();
    }

    // <summary>
    // This is an actual command. It implements the Command interface
    // and it defines the binding between the receiver object and its
    // action. It implements the execute method by calling the corresponding
    // action on the receiver.
    // </summary>
    class ConcreteCommand : Command
    {
        private Receiver _receiver;

        public ConcreteCommand(Receiver receiver)
        {
            this._receiver = receiver;
        }

        public void Execute()
        {
            this._receiver.Action();
        }
    }

    // <summary>
    // The invoker asks the command to carry out the request.
    // It is bound to one specific command at any given time.
    // </summary>
    class Invoker
    {
        private Command _command;

        public void SetCommand(Command command)
        {
            this._command = command;
        }

        public void ExecuteCommand()
        {
            this._command.Execute();
        }
    }
}
```

## Resources

- Wikipedia
- OODesign
- DoFactory