Command Pattern

# Command Pattern is to use one single ICommand interface to manage different types of concrete commands for different operations, collection of commands and Undo Redo etc.

# ICommand interface defines all the operations a command may have. E.g. Do(), Undo(), Redo(), Insert(), Show() etc.

# While concrete commands could be InfoCommand() where its show() displays Green and WarningCommand() where its show() displays Red which are all inherited from ICommand interface

# Class UML Diagram

Diagram

Description automatically generated

# Sample code

Reference Link: <https://refactoring.guru/design-patterns/command/csharp/example>

**Usage examples:**

The Command pattern is pretty common in C# code. Most often it’s used as an alternative for callbacks to parameterizing UI elements with actions. It’s also used for queueing tasks, tracking operations history, etc.

**Identification:**

The Command pattern is recognizable by behavioral methods in an abstract/interface type (sender) which invokes a method in an implementation of a different abstract/interface type (receiver) which has been encapsulated by the command implementation during its creation. Command classes are usually limited to specific actions.

**Conceptual Example**

This example illustrates the structure of the Command design pattern. It focuses on answering these questions:

What classes does it consist of?

What roles do these classes play?

In what way the elements of the pattern are related?

Program.cs: Conceptual example

using System;

namespace RefactoringGuru.DesignPatterns.Command.Conceptual

{

// The Command interface declares a method for executing a command.

public interface ICommand

{

void Execute();

}

// Some commands can implement simple operations on their own.

class SimpleCommand : ICommand

{

private string \_payload = string.Empty;

public SimpleCommand(string payload)

{

this.\_payload = payload;

}

public void Execute()

{

Console.WriteLine($"SimpleCommand: See, I can do simple things like printing ({this.\_payload})");

}

}

// However, some commands can delegate more complex operations to other

// objects, called "receivers."

class ComplexCommand : ICommand

{

private Receiver \_receiver;

// Context data, required for launching the receiver's methods.

private string \_a;

private string \_b;

// Complex commands can accept one or several receiver objects along

// with any context data via the constructor.

public ComplexCommand(Receiver receiver, string a, string b)

{

this.\_receiver = receiver;

this.\_a = a;

this.\_b = b;

}

// Commands can delegate to any methods of a receiver.

public void Execute()

{

Console.WriteLine("ComplexCommand: Complex stuff should be done by a receiver object.");

this.\_receiver.DoSomething(this.\_a);

this.\_receiver.DoSomethingElse(this.\_b);

}

}

// The Receiver classes contain some important business logic. They know how

// to perform all kinds of operations, associated with carrying out a

// request. In fact, any class may serve as a Receiver.

class Receiver

{

public void DoSomething(string a)

{

Console.WriteLine($"Receiver: Working on ({a}.)");

}

public void DoSomethingElse(string b)

{

Console.WriteLine($"Receiver: Also working on ({b}.)");

}

}

// The Invoker is associated with one or several commands. It sends a

// request to the command.

class Invoker

{

private ICommand \_onStart;

private ICommand \_onFinish;

// Initialize commands.

public void SetOnStart(ICommand command)

{

this.\_onStart = command;

}

public void SetOnFinish(ICommand command)

{

this.\_onFinish = command;

}

// The Invoker does not depend on concrete command or receiver classes.

// The Invoker passes a request to a receiver indirectly, by executing a

// command.

public void DoSomethingImportant()

{

Console.WriteLine("Invoker: Does anybody want something done before I begin?");

if (this.\_onStart is ICommand)

{

this.\_onStart.Execute();

}

Console.WriteLine("Invoker: ...doing something really important...");

Console.WriteLine("Invoker: Does anybody want something done after I finish?");

if (this.\_onFinish is ICommand)

{

this.\_onFinish.Execute();

}

}

}

class Program

{

static void Main(string[] args)

{

// The client code can parameterize an invoker with any commands.

Invoker invoker = new Invoker();

invoker.SetOnStart(new SimpleCommand("Say Hi!"));

Receiver receiver = new Receiver();

invoker.SetOnFinish(new ComplexCommand(receiver, "Send email", "Save report"));

invoker.DoSomethingImportant();

}

}

}

Output.txt: Execution result

Invoker: Does anybody want something done before I begin?

SimpleCommand: See, I can do simple things like printing (Say Hi!)

Invoker: ...doing something really important...

Invoker: Does anybody want something done after I finish?

ComplexCommand: Complex stuff should be done by a receiver object.

Receiver: Working on (Send email.)

Receiver: Also working on (Save report.)