# How to use the command design pattern in C#

Design patterns are proven solutions used to solve common design problems and reduce the complexities in code. The Gang of Four design patterns fall into three categories:

* Creational — patterns related to object creation
* Structural — patterns related to object assembly
* Behavioral — patterns related to object collaboration and separation of responsibilities

The command design pattern falls under the behavioral pattern category. This article explores how we can work with the command design pattern in C#.

## What is the command design pattern?

The intent of the command design pattern is to decouple the requester of an action from the object that executes the action. In the command design pattern, a request is encapsulated as an object that contains all information about the request. This object is then passed to an invoker object. The invoker object then looks for the appropriate object to handle the command and passes the command to the object.

The command design pattern is a good choice when you want to implement callbacks, queuing tasks, tracking history, and undo/redo functionality in your application. The command pattern is a good choice for implementing retry mechanisms - when your application would like to reattempt connecting to a service at a later point in time which is not up and running at this moment. The Command pattern is also used in message queuing applications, i.e., in applications that need to recover from data loss.

## Command design pattern participants

In a classic implementation of the Command pattern you have four components: the command, the invoker, the receiver, and the client. The participants in the command design pattern include the following:

* Command — provides an interface for executing an operation
* ConcreteCommand — extends the Command interface and implements the Execute method
* Client — instantiates a ConcreteCommand class
* Invoker — informs the command to execute the request
* Receiver — contains the logic for executing the operations associated with the request

## Command design pattern example in C#

In the next section we’ll explore how we can implement the command design pattern. In our example, we will implement a simple calculator using the following classes:

* Command (Command abstract base class)
* SimpleCalculator (Receiver class)
* AddCommand (concrete command class)
* SubstractCommand (concrete command class)
* Multiply Command (concrete command class)
* DivideCommand (concrete command class)
* Invoker (Invoker class)

## Create the Command abstract base class in C#

Consider the following abstract base class named Command that contains the declaration of the Execute method.

**public** **abstract** **class** **Command**  
    {  
        **protected** **SimpleCalculator** receiver;  
        **public** **Command**(**SimpleCalculator** receiver)  
        {  
            **this**.receiver = receiver;  
        }  
        **public** **abstract** **int** **Execute**();  
    }

The following enum shows the operations that will be supported in our simple calculator.

**public** **enum** **CommandOption**  
    {  
        **Add**, **Substract**, **Multiply**, **Divide**  
    }

## Create the Receiver class in C#

The following is a class named SimpleCalculator. This class acts as the Receiver and contains the definition of the Add, Subtract, Multiply, and Divide methods.

**public** **class** **SimpleCalculator**  
    {  
        **private** **int** \_x, \_y;  
        **public** **SimpleCalculator**(**int** a, **int** b)  
        {  
            \_x = a;  
            \_y = b;  
        }  
        **public** **int** **Add**()  
        {  
            **return** \_x + \_y;  
        }  
        **public** **int** **Subtract**()  
        {  
            **return** \_x - \_y;  
        }  
        **public** **int** **Multiply**()  
        {  
            **return** \_x \* \_y;  
        }  
        **public** **int** **Divide**()  
        {  
            **return** \_x / \_y;  
        }  
    }

## Create the concrete command classes in C#

The concrete command classes extend the Command abstract base class and implement the Execute method as shown below.

**public** **class** **AddCommand** : **Command**  
    {  
        **private** **SimpleCalculator** \_calculator;  
        **public** **AddCommand**(**SimpleCalculator** calculator) : **base**(calculator)  
        {  
            \_calculator = calculator;  
        }  
        **public** **override** **int** **Execute**()  
        {  
            **return** \_calculator.**Add**();  
        }  
    }  
    **public** **class** **SubtractCommand** : **Command**  
    {  
        **private** **SimpleCalculator** \_calculator;  
        **public** **SubtractCommand**(**SimpleCalculator** calculator) :  
        **base**(calculator)  
        {  
            \_calculator = calculator;  
        }  
        **public** **override** **int** **Execute**()  
        {  
            **return** \_calculator.**Subtract**();  
        }  
    }  
    **public** **class** **MultiplyCommand** : **Command**  
    {  
        **private** **SimpleCalculator** \_calculator;  
        **public** **MultiplyCommand**(**SimpleCalculator** calculator) :  
        **base**(calculator)  
        {  
            \_calculator = calculator;  
        }  
        **public** **override** **int** **Execute**()  
        {  
            **return** \_calculator.**Multiply**();  
        }  
    }  
    **public** **class** **DivideCommand** : **Command**  
    {  
        **private** **SimpleCalculator** \_calculator;  
        **public** **DivideCommand**(**SimpleCalculator** calculator) :  
        **base**(calculator)  
        {  
            \_calculator = calculator;  
        }  
        **public** **override** **int** **Execute**()  
        {  
            **return** \_calculator.**Divide**();  
        }  
    }

## Create the Invoker class in C#

The following code snippet illustrates the Invoker class. It contains two methods, SetCommand and Execute. While SetCommand is used to assign the command object to the private Command reference in the Invoker class, Execute is used to execute the command.

**public** **class** **Invoker**  
    {  
        **private** **Command** \_command;  
        **public** **void** **SetCommand**(**Command** command)  
        {  
            \_command = command;  
        }  
        **public** **int** **Execute**()  
        {  
            **return** \_command.**Execute**();  
        }  
    }

## The command design pattern in action in C#

Finally, the following code snippet illustrates how you can perform a simple calculation using the SimpleCalculator class.

**static** **void** **Main**(**string**[] args)  
        {  
            **SimpleCalculator** calculator = **new** **SimpleCalculator**(15, 3);  
            **var** addCommand = **new** **AddCommand**(calculator);  
            **var** substractCommand = **new** **SubtractCommand**(calculator);  
            **var** multiplyCommand = **new** **MultiplyCommand**(calculator);  
            **var** divideCommand = **new** **DivideCommand**(calculator);  
            **Invoker** invoker = **new** **Invoker**();  
            invoker.**SetCommand**(addCommand);  
            **Console**.**WriteLine**("Result is {0}", invoker.**Execute**());  
            invoker.**SetCommand**(substractCommand);  
            **Console**.**WriteLine**("Result is {0}", invoker.**Execute**());  
            invoker.**SetCommand**(multiplyCommand);  
            **Console**.**WriteLine**("Result is {0}", invoker.**Execute**());  
            invoker.**SetCommand**(divideCommand);  
            **Console**.**WriteLine**("Result is {0}", invoker.**Execute**());  
            **Console**.**ReadLine**();  
        }

The command design pattern provides support for extensibility and reduces the coupling that exists between the invoker and receiver of a command. Since the request is encapsulated into a stand-alone object, you can parameterize methods with different requests, save requests in a queue, and even provide support for redo-able or undo-able operations.