37. Command

Command is a design pattern that isolates an operation in its own object. This makes such an operation reusable, so you don't have to construct a brand new logic every time you need to perform such an operation.

The most popular way of using Command is in database operations. A Command object will contain all necessary information to perform some action that would modify the data (insertion, update, deletion, etc.). Then, if you need to perform such an operation on a particular database table, all you have to do is just reuse this object and maybe modify some of its parameters. You won't have to write any complex logic involving object-relationship mappers or direct database queries.

Command can be summarized as follows:

- There is a **Command** interface that describes the behavior that each command should have. Usually, it only has one method, which executes the action.
- If multiple related commands are needed, it makes sense to add **Command** abstract class with the shared functionality.
- Each **Concrete Command** accepts parameters in its constructor and implements the method that executes the action.
- Because **Command** accepts constructor parameters, it's intended to be a single-use object.
- Optionally, there could be an **Invoker** object, the role of which is to decide which commands to execute.

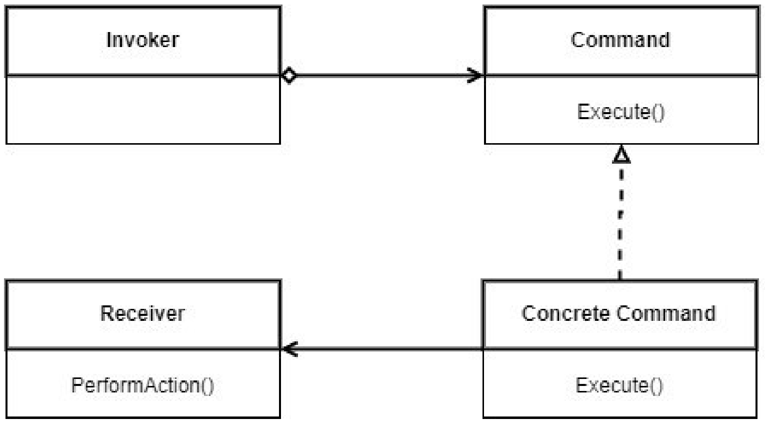


Figure 37.1 - Command UML diagram

We will now go through an example implementation of Command. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Behavioral_Patterns/Command

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Command implementation example

As in all other examples, we will create a console application project. And the first thing we will do in this project is add the interface for our **Command** object, which will be as follows:

```
namespace Command_Demo;

internal interface ICommand
{
    void Execute();
}
```

As you can see, the only method we have is `Execute`. In our example, it's blocking `void`. But it could be an asynchronous `Task` where appropriate. **Command** is meant to execute something rather than return data. This is why there is no return value in our example.

Next, we will add `DataReceiver.cs` file with the following content:

```
namespace Command_Demo;

internal class DataReceiver
{
    private readonly
        Dictionary<string,string> data;

    public DataReceiver()
    {
        data = [];
    }

    public void Upsert(
        string key, string value)
    {
        data[key] = value;
        Console.WriteLine(
            $"Upserted: {key} - {value}.");
    }

    public void Delete(string key)
    {
        data.Remove(key);
        Console.WriteLine(
            $"Removed: {key}.");
    }
}
```

This file mimics a database access layer. You can imagine that, instead of dealing with in-memory `Dictionary`, you are working with a database table.

Next, we will add the following abstract class. We will need this, because all of our commands will depend on `DataReceiver` we've created earlier. So we put it into the constructor.

```
namespace Command_Demo;

internal abstract class Command(
    DataReceiver receiver) : ICommand
{
    protected DataReceiver
        receiver = receiver;

    public abstract void Execute();
}
```

Next, we will add the following **Concrete Command** instance, which performs upsertion (insertion or update) of values into the `DataReceiver`.

```csharp
namespace Command_Demo;

internal class UpsertCommand : Command
{
    private readonly string key;
    private readonly string value;

    public UpsertCommand(string key,
        string value,
        DataReceiver receiver) : base(receiver)
    {
        this.key = key;
        this.value = value;
    }

    public override void Execute()
    {
        receiver.Upsert(key, value);
    }
}
```

Then, we will add the following **Concrete Command**, which deletes a specified entry from the `DataReceiver` based on its key.

```csharp
namespace Command_Demo;

internal class DeleteCommand(
    string key,
    DataReceiver receiver) :
    Command(receiver)
{
    public override void Execute()
    {
        receiver.Delete(key);
    }
}
```

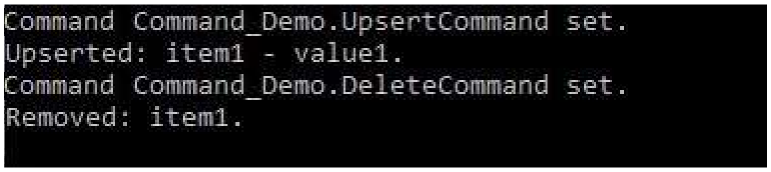
In our example, we will also add a command **Invoker**. It's not strictly necessary, as you can use **Command** instance directly by the calling code. But it's still useful for demonstration purposes.

```
namespace Command_Demo;

internal class DataCommandInvoker
{
    private ICommand? command;

    public void SetCommand(
        ICommand command)
    {
        this.command = command;
        Console.WriteLine(
            $"Command {command.GetType()
            } set.");
    }

    public void ExecuteCommand()
    {
        command?.Execute();
    }
}
```

Finally, we will replace the content of `Program.cs` file with the following:

```
using Command_Demo;

var dataReceiver =
    new DataReceiver();
var invoker =
    new DataCommandInvoker();
invoker.SetCommand(
    new UpsertCommand(
        "item1", "value1",
        dataReceiver));
invoker.ExecuteCommand();
invoker.SetCommand(
    new DeleteCommand(
        "item1",
        dataReceiver));
invoker.ExecuteCommand();
Console.ReadKey();
```

Essentially, what we are doing here. is inserting an item into our data collection and then deleting this item from it. And this is what the output shows:



```
Command Command_Demo.UpsertCommand set.
Upserted: item1 - value1.
Command Command_Demo.DeleteCommand set.
Removed: item1.
```

Figure 37.2 - Commands successfully executed

And this concludes the overview of Command design pattern. Let's summarize its main benefits.

Benefits of using Command

Command design pattern has the following benefits:

- Single responsibility principle is well maintained, as the operation is fully separated from the objects it operates on.
- Open-closed principle is well maintained, as the operation is added in a separate class without modifying any existing classes.
- You can easily reuse a **Command** implementation if you want to perform a number of similar operations.
- You can combine it with Chain of Responsibility and assemble multiple simple commands into one complex command.
- You can combine it with Memento and enable undo/redo actions.
- Because the entire process of execution is controlled within a **Command** implementation, you can fully control the execution logic, including any delays to it.

Let's now have a look at things to look out for while using the Command design pattern.

Caveats of using Command

Perhaps the main thing you have to look out for while using Command is that you can accidentally make your code too complicated if you are not careful enough.

First of all, you may end up with too many **Command** objects that will become hard to manage. However, it might be that not every operation that you need to perform warrants the use of Command. For example, if an operation is definitely neither customizable nor reusable, it may not be needed to be isolated into its own **Command** object.

Secondly, if you make any particular **Command** implementation too customizable, you may end up with a badly written object that is trying to do too many things at once and violates single responsibility principle as the result. So, it's OK to have some customization in a **Command** implementation. But make sure it's only some relatively minor customization.

Remember that Command, just like any other design patterns, is meant to make your code more maintainable and not less maintainable.