



## Command pattern

---

Bei der **objektorientierten Programmierung** ist das **Command Pattern** (**Befehlsmuster**) ein **Verhaltensentwurfsmuster**, bei dem ein Objekt verwendet wird, um alle Informationen zu **kapseln**, die zum Ausführen einer Aktion oder zum Auslösen eines Ereignisses zu einem späteren Zeitpunkt erforderlich sind. Diese Informationen umfassen den Methodennamen, das Objekt, dem die Methode gehört, und Werte für die Methodenparameter.

Vier Begriffe, die immer mit dem Befehlsmuster sind, verbunden sind **Befehl**, **Empfänger**, **Aufrufer** und **Client**.

Ein **Befehl** Objekt kennt **Empfänger** und ruft eine **Methode** des **Empfängers**.

Werte für Parameter der Empfänger methode werden im Befehl gespeichert. Das Empfängerobjekt zum Ausführen dieser Methoden wird ebenfalls durch **Aggregation** im Befehlsobjekt gespeichert.

Der **Empfänger** dann macht die Arbeit, wenn das `execute()` Verfahren in **Befehl** aufgerufen wird.

Ein **Anruferobject** weiß, wie ein **Befehl** ausgeführt wird, und führt optional eine Buchhaltung über die Befehlsausführung durch.

Der **Aufrufer** weiß nichts über einen konkreten **Befehl**, es kennt nur über die **Befehlsschnittstelle**. **Aufruferobjekte**, **Befehlsobjekte** und **Empfängerobjekte** werden von einem **Clientobject** gehalten, der **Client** entscheidet, welche **Empfängerobjekte** er den **Befehlsobjekten** zuweist und welche **Befehle** er dem **Aufrufer** zuweist.

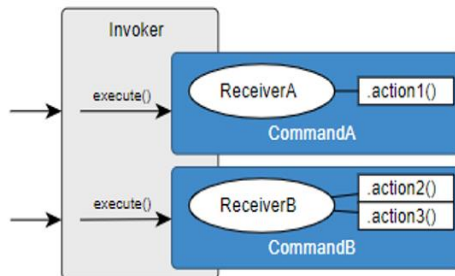
Der **Client** entscheidet, welche **Befehle** an welchen Punkten ausgeführt werden sollen. Um einen **Befehl** auszuführen, wird das **Befehlsobjekt** an das **Aufruferobjekt** übergeben.

Die Verwendung von **Befehlsobjekten** erleichtert die Erstellung allgemeiner Komponenten, die Methodenaufrufe zu einem Zeitpunkt ihrer Wahl delegieren, sequenzieren oder ausführen

müssen, ohne die Klasse der Methode oder die Methodenparameter kennen zu müssen. Die Verwendung eines **Aufruferobjekts** ermöglicht die bequeme Durchführung der Buchhaltung über Befehlsausführungen sowie die Implementierung verschiedener Modi für Befehle, die vom **Aufruferobjekt** verwaltet werden, ohne dass der **Client** über das Vorhandensein von Buchhaltung oder Modi informiert sein muss.

Die zentralen Ideen dieses Entwurfsmusters spiegeln die Semantik [erstklassiger Funktionen](#) und [Funktionen höherer Ordnung](#) in [funktionalen Programmiersprachen sehr gut wider](#). Insbesondere ist das Aufruferobjekt eine Funktion höherer Ordnung, deren Befehlsobjekt ein erstklassiges Argument ist.

#### Command



Das Command Design Pattern ermöglicht die Modularisierung von Befehlen und Aufrufen. Auf elegante Weise können Befehle rückgängig gemacht, protokolliert oder in einer Warteschlange gelegt werden. [Mehr zum Command Pattern... >>](#)

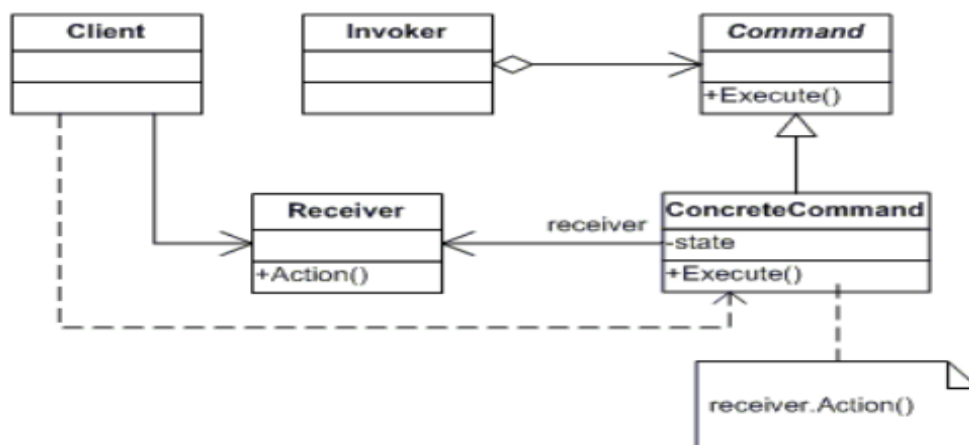
## Übersicht

Die Verwendung des Befehlsmusters kann folgende Probleme lösen: [\[2\]](#)

- Das Koppeln des Aufrufers einer Anfrage an eine bestimmte Anfrage sollte vermieden werden. Das heißt, fest verdrahtete Anforderungen sollten vermieden werden.
- Es sollte möglich sein, ein Objekt (das eine Anforderung aufruft) mit einer Anforderung zu konfigurieren.

Auf diese Weise kann eine Klasse mit einem Befehlsobjekt konfiguriert werden, mit dem eine Anforderung ausgeführt wird. Die Klasse ist nicht mehr an eine bestimmte Anforderung gekoppelt und hat keine Kenntnis (ist unabhängig davon), wie die Anforderung ausgeführt wird.

## UML-Klassendiagramm



## Teile des Befehlsmusters

Das Befehlsmuster ist in fünf Teile unterteilt:

- Der **Befehl** deklariert eine Schnittstelle für die Ausführung einer Operation.

- Der **ConcreteCommand** definiert eine Bindung zwischen einem **Empfänger** und einer Aktion.
- Der **Client** erstellt ein ConcreteCommand-Objekt und legt einen **Empfänger** für den Befehl fest.
- Der **Invoker** fordert den Befehl auf, seine Anfrage auszuführen.
- Der **Empfänger** weiß, wie die mit der Aktion der Anforderung verbundenen Operationen ausgeführt werden.

## Teilnehmer

Die Klassen und Objekte, die an diesem Muster teilnehmen, sind:

- **Befehl** ( **Command**)
  - deklariert eine Schnittstelle zum Ausführen einer Operation
- **ConcreteCommand** ( **CalculatorCommand**)
  - Definiert eine Bindung zwischen einem Empfängerobjekt und einer Aktion
  - implementiert Execute durch Aufrufen der entsprechenden Operation (en) auf dem Empfänger
- **Client** ( **CommandApp**)
  - Erstellt ein ConcreteCommand-Objekt und legt dessen Empfänger fest
- **Invoker** ( **User**)
  - fordert den Befehl auf, die Anforderung auszuführen
- **Empfänger** ( **Calculator**)
  - weiß, wie die mit der Ausführung der Anforderung verbundenen Vorgänge ausgeführt werden.

## Beispiel

// Empfänger A (Funktionen- Aktions)

```
public class ReceiverA
{
    public void action1()
    {
        Console.WriteLine("ReceiverA.action1()");
    }
    public void action2()
    {
        Console.WriteLine("ReceiverA.action2()");
    }
    public void action3()
    {
        Console.WriteLine("ReceiverA.action3()");
    }
    //weitere Methoden...
}
```

// Empfänger B (Funktionen- Aktions)

```
public class ReceiverB
{
    public void action1()
    {
        Console.WriteLine("ReceiverB.action1()");
    }
    public void action2()
    {
        Console.WriteLine("ReceiverB.action2()");
    }
    public void action3()
    {
        Console.WriteLine("ReceiverB.action3()");
    }
    //weitere Methoden...
}
```

\_\_\_\_\_Interface \_\_\_\_\_

```
interface Command
{
    public void execute();
}
```

\_\_\_\_\_ Befehl A \_\_\_\_\_

```
class BefehlA : Command
{
    private ReceiverA receiverA;

    public BefehlA(ReceiverA receiver)
    {
        this.receiverA = receiver;
    }

    public void execute()
    {
        receiverA.action1();
        receiverA.action3();
        receiverA.action3();
    }
}
```

\_\_\_\_\_ Befehl B \_\_\_\_\_

```
public class BefehlB :Command
{
    public ReceiverB receiverB;

    public BefehlB(ReceiverB receiver)
    {
        this.receiverB = receiver;
    }

    public void execute()
    {
        receiverB.action1();
        receiverB.action2();
        receiverB.action3();
    }
}
```

```

class Invoker
{
    //Methode, um Commands zu setzen....
    private Command com1; //default möglich
    private Command com2; //default möglich

    //Eine Variante: Methode, die für Ausführung des Befehls sorgt.
    //kann z. B. durch ein Event ausgelöst werden
    public void doCom1()
    {
        // com1 = new BefehlA();
        com1.execute();
    }
    public void doCom2()
    {
        com2.execute();
    }

    //Methode, mit der der Invoker konfiguriert werden kann
    //bzw. mit der die Commands geladen werden
    public void setCom1(Command command1)
    {
        this.com1 = command1;
    }

    public void setCom2(Command command2)
    {
        this.com2 = command2;
    }
}

```

---

#### Main

---

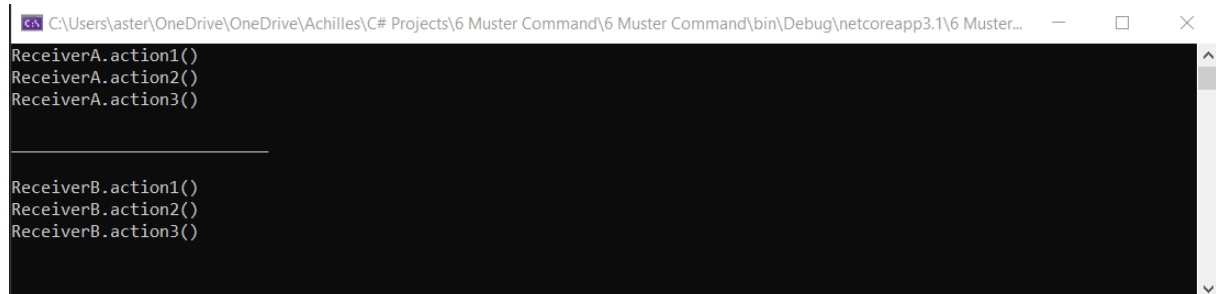
```

static void Main(string[] args)
{
    ReceiverA receiverA = new ReceiverA();
    ReceiverB receiverB = new ReceiverB();
    Command befehlA = new BefehlA(receiverA);
    Command befehlB = new BefehlB(receiverB);
    Invoker invoker = new Invoker();

    invoker.setCom1(befehlA);
    invoker.doCom1();
    Console.WriteLine("\n_____ \n");
    invoker.setCom2(befehlB);
    invoker.doCom2();
    Console.ReadKey();
}

```

#### Output



```

ReceiverA.action1()
ReceiverA.action2()
ReceiverA.action3()
-----
ReceiverB.action1()
ReceiverB.action2()
ReceiverB.action3()

```