

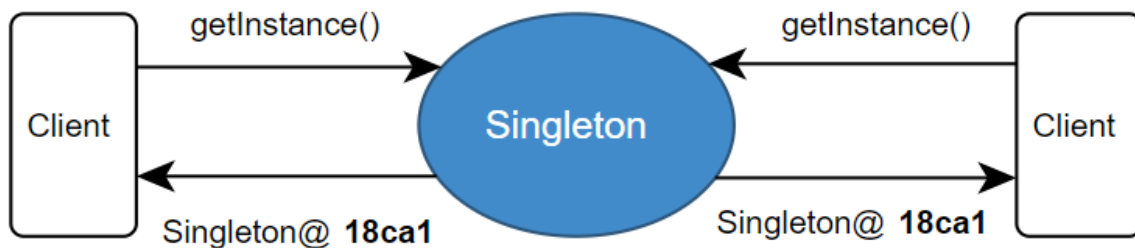
# DESIGN-PATTERN

## SINGLETON

### Singleton (Entwurfsmuster)

---

Das **Singleton** (selten auch **Einzelstück** genannt) ist ein in der [Softwareentwicklung](#) eingesetztes [Entwurfsmuster](#) und gehört zur Kategorie der [Erzeugungsmuster](#) (engl. *creational patterns*). Es stellt sicher, dass von einer Klasse genau ein Objekt existiert.<sup>[1]</sup> Dieses Singleton ist darüber hinaus üblicherweise global verfügbar. Das Muster ist eines der von der sogenannten [Viererbande](#) (GoF) publizierten Muster.



### Verwendung

---

Das Singleton findet Verwendung, wenn

- nur ein [Objekt](#) zu einer [Klasse](#) existieren darf und ein einfacher Zugriff auf dieses Objekt benötigt wird oder
- das einzige Objekt durch Unterklassenbildung spezialisiert werden soll.

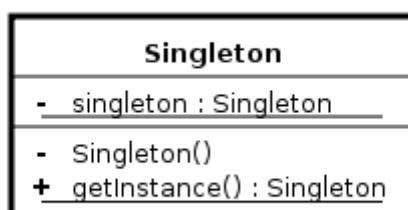
Anwendungsbeispiele sind

- ein zentrales Protokoll-Objekt, das Ausgaben in eine [Datei](#) schreibt.
- Druckaufträge, die zu einem [Drucker](#) gesendet werden, sollen nur in einen einzigen [Puffer](#) geschrieben werden.

### UML-Diagramm

---

Siehe auch: [UML](#)



## Eigenschaften

---

Das Einzelstück (*Singleton*)

- erzeugt und verwaltet das einzige Objekt der Klasse
- bietet globalen Zugriff auf dieses Objekt über eine Instanzoperation (*getInstance()*).

Dabei ist

- die Instanzoperation eine Klassenmethode, das heißt [statisch gebunden](#)
- das private Attribut „Instanz“ (*singleton*) ein Klassenattribut, das heißt ein statisches Attribut.

*In Klammern stehen die Bezeichnungen aus obiger Abbildung.*

## Vorteile

---

Das Muster bietet eine Verbesserung gegenüber globalen Variablen:

- Zugriffskontrolle kann realisiert werden.
- Das Singleton kann durch Unterklassenbildung spezialisiert werden.
- Welche Unterklasse verwendet werden soll, kann zur Laufzeit entschieden werden.
- Die Einzelinstanz muss nur erzeugt werden, wenn sie benötigt wird.
- Sollten später mehrere Objekte benötigt werden, ist eine Änderung leichter möglich als bei globalen Variablen.

## Nachteile

---

- Es besteht die große Gefahr, durch exzessive Verwendung von Singletons quasi ein Äquivalent zu [globalen Variablen](#) zu implementieren und damit dann prozedural anstatt objektorientiert zu programmieren.<sup>[2]</sup>
- Abhängigkeiten zur Singleton-Klasse werden verschleiert, d. h. ob eine Singleton-Klasse verwendet wird, erschließt sich nicht aus dem Interface einer Klasse, sondern nur anhand der Implementierung.<sup>[3]</sup> Zudem wird die [Kopplung](#) erhöht, was Wiederverwendbarkeit und Übersichtlichkeit einschränkt.
- Eine Ressourcen-Deallokation von Ressourcen, die das Singleton verwendet, ist schwierig. So ist zum Beispiel bei einem Singleton für ein Logging-System oft unklar, wann die Logdatei geschlossen werden soll.

Wegen der vielen Nachteile wird das Singleton-Muster (und auch das Idiom *Double-checked Locking*) mitunter schon als [Anti-Pattern](#) bewertet. Für Fälle, in denen tatsächlich technisch ein passender Bereich für ein Singleton existiert, können Singletons aber sinnvoll sein – insbesondere wenn sie sich auf andere „einmalige Strukturen“ wie zum Beispiel eine [Abstract Factory](#) beziehen. Trotzdem: Das korrekte Design von Singletons ist schwierig – in der Regel schwieriger als Designs ohne Singletons.

## Verwendung in der Analyse

---

In der Analyse wird ein (fachliches) Singleton in der Regel dadurch gekennzeichnet, dass die Multiplizität der Klasse als 1 definiert wird. Wie auch im Design muss der Bereich der Multiplizität hinterfragt werden: Gibt es tatsächlich nur „eine Zentralstelle für ...“, oder können zum Beispiel in länderübergreifenden Systemen sehr wohl mehrere Objekte einer Sorte existieren?

## Implementierung

### Verwandte Entwurfsmuster

---

Die Eigenschaften des Singleton treffen für viele Klassen der anderen Muster zu, so dass diese dann als Singleton ausgeführt werden.

Zum Beispiel sind [abstrakte Fabriken](#), [Erbauer](#) oder [Prototypen](#) oft auch Singleton.

## BEISPIEL

```
class Ork_General
{
    public string name;
    private static Ork_General My_General;
    private Ork_General()
    {

    }
    public static Ork_General getInstance()
    {
        if (My_General == null)
        {
            My_General = new Ork_General();
        }
        return My_General;
    }
    public void setName(string General)
    {
        name = General;
    }
    public string AngriffsBefell()
    {
        return name + " Befehl : Jetzt greifen wir an !!!!!";
    }
    public string RückzugsBefell()
    {
        return name + " Befehl : Jetzt ziehen wir zurück !!!!!";
    }
}
```

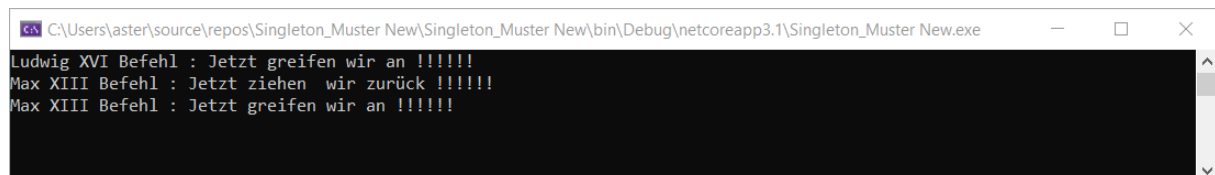
---

### MAIN

---

```
static void Main(string[] args)
{
    Ork_General King = Ork_General.getInstance();
    King.setName("Ludwig XVI");
    Console.WriteLine(King.AngriffsBefell());

    Ork_General King2 = Ork_General.getInstance();
    King2.setName("Max XIII");
    Console.WriteLine(King2.RückzugsBefell());
    Console.WriteLine(King.AngriffsBefell());
    Console.ReadKey();
}
```



The screenshot shows a Windows command prompt window titled "C:\Users\aster\source\repos\Singleton\_Muster New\Singleton\_Muster New\bin\Debug\netcoreapp3.1\Singleton\_Muster New.exe". The output of the program is displayed as follows:

```
Ludwig XVI Befehl : Jetzt greifen wir an !!!!!
Max XIII Befehl : Jetzt ziehen wir zurück !!!!!
Max XIII Befehl : Jetzt greifen wir an !!!!!
```