

**Objektorientiert mit C:**

**Typische UML-Diagramme und ihre Umsetzung**

Frank Listing  
f.listing@microconsult.com

- Wie sieht die aktuelle Situation in der SW-Entwicklung aus?
- Welche UML-Elemente können in Code umgesetzt werden?
- Wie können objektorientierte Elemente in C umgesetzt werden?

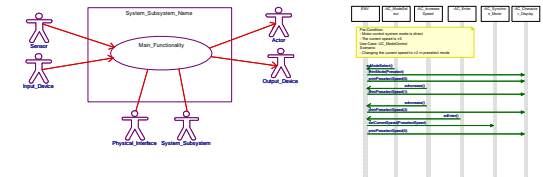
---

## Aktuelle Situation

---

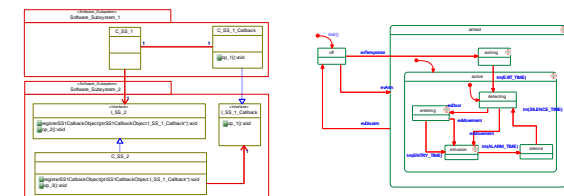
## SW-Entwicklung heute:

- Immer **größere** und **komplexere Projekte** erfordern neue SW-Techniken.
- Um die Komplexität zu beherrschen, wird immer mehr **objektorientiert** entwickelt.
- Die **einheitliche Notation** ist die **UML**.



## Deshalb Einsatz der UML für:

- Beschreibung der **Anforderungen**
- Beschreibung der **Architektur**
- Beschreibung des **Designs**



Auch im der **embedded Entwicklung** wird die **UML** vermehrt zur Beschreibung der Architektur und des Verhaltens von Applikationen eingesetzt.

Auf die objektorientierte Programmierung wird aber häufig **verzichtet**, da **C nicht objektorientiert** ist.

# Kann mit C wirklich nicht objektorientiert entwickelt werden?

# Doch!



- Objektorientierung findet im Kopf statt, nicht in der Programmiersprache.
- C bietet mehr Möglichkeiten als geahnt.
- Regeln und Programmiervorschriften können genutzt werden.

---

# Implementierung von UML-Elementen

---

Viele Elemente der UML sind völlig unabhängig von der Programmiersprache.

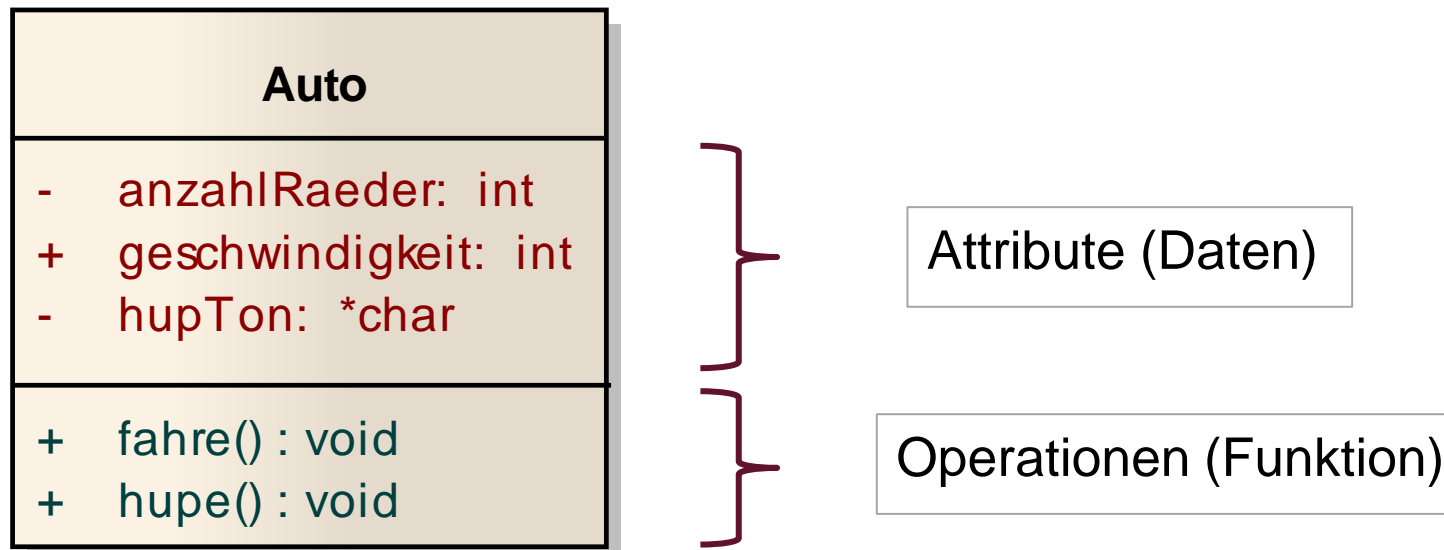
- Anforderungen
- Pakete
- zeitliche Abläufe

Folgende Elemente aus dem UML-Klassendiagramm werden am häufigsten im Code umgesetzt:

- Klassen
- Beziehungen
  - Assoziation
  - Komposition (Aggregation)
  - Generalisierung (Vererbung)

Eines der wichtigsten Elemente in der UML ist die **Klasse**. Sie steht im Mittelpunkt der objektorientierten Programmierung.

In der Programmiersprache C gibt es kein Syntaxelement für die Klasse. Allerdings gibt es das **struct**-Element als Mittel, komplexe Daten strukturiert abzulegen.



Eine **Struktur** mit zugeordneten Funktionen wird definiert.  
Der Zugriff auf die Struktur erfolgt nur über **spezielle Funktionen**.

```
typedef struct
{
    int anzahlRaeder;
    char* hupTon;
    int geschwindigkeit;
}Auto;

void Auto_init(Auto *self,
               int anzahlRaeder, char* hupTon);

void Auto_fahre(Auto *self);

void Auto_hupe(Auto *self, int anzahl);
```

Attribute (Daten)

Operationen (Funktion)

Zuordnung von Funktion zur Struktur

Objekte können auf dem **Stack** oder dynamisch auf dem **Heap** angelegt werden.

```
// Objekt auf dem Stack
Auto a;

Auto_init(&a, 4, "tut");
Auto_fahre(&a);
Auto_hupe(&a, 3);
```

Dank `malloc()` ist das dynamische Anlegen von Objekten auf dem Heap kein Problem.

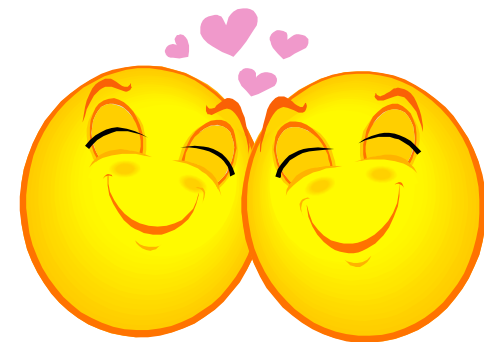
```
// Objekt auf dem Heap
Auto *pa= (Auto*)malloc(sizeof(Auto));

Auto_init(pa, 3, "maep");
Auto_fahre(pa);
Auto_hupe(pa, 2);

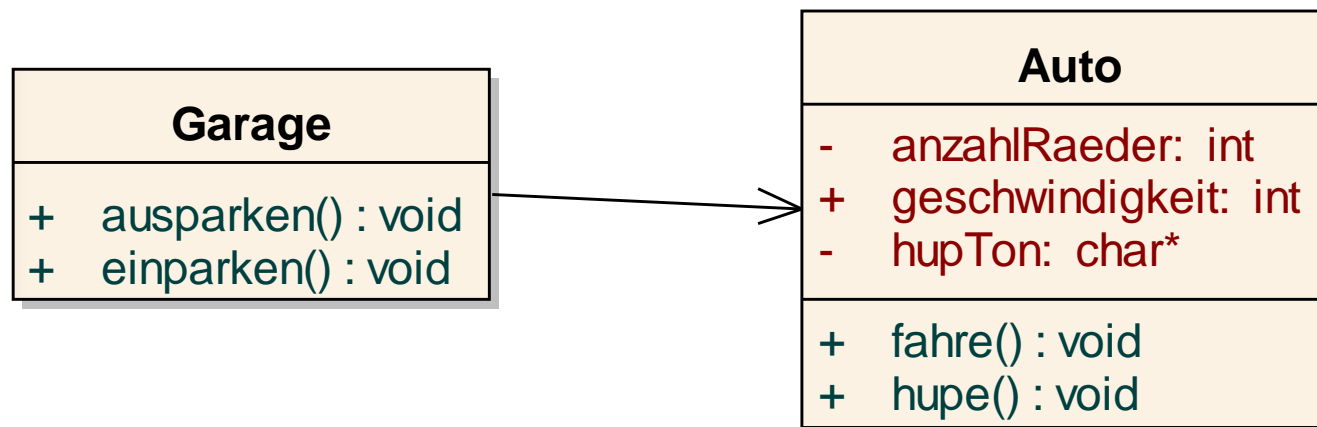
free(pa);
```

Beziehungen beschreiben die Zusammenarbeit zwischen einzelnen Klassen.

- Klassen können sich kennen.
- Eine Klasse kann andere Klassen beinhalten.
- Eine Klasse kann Eigenschaften an eine andere Klasse weitergeben.



Die **Assoziation** ist eine einfache Beziehung zwischen zwei gleichrangigen Klassen.



Eine Klasse enthält einen **Zeiger** auf die andere Klasse.

```
typedef struct
{
    Auto* pAuto;
}Garage;

void Garage_init(Garage *self);
void Garage_init2(Garage *self, Auto *pAuto);
void Garage_einparken(Garage *self, Auto *pAuto);
void Garage_ausparken(Garage *self);
```

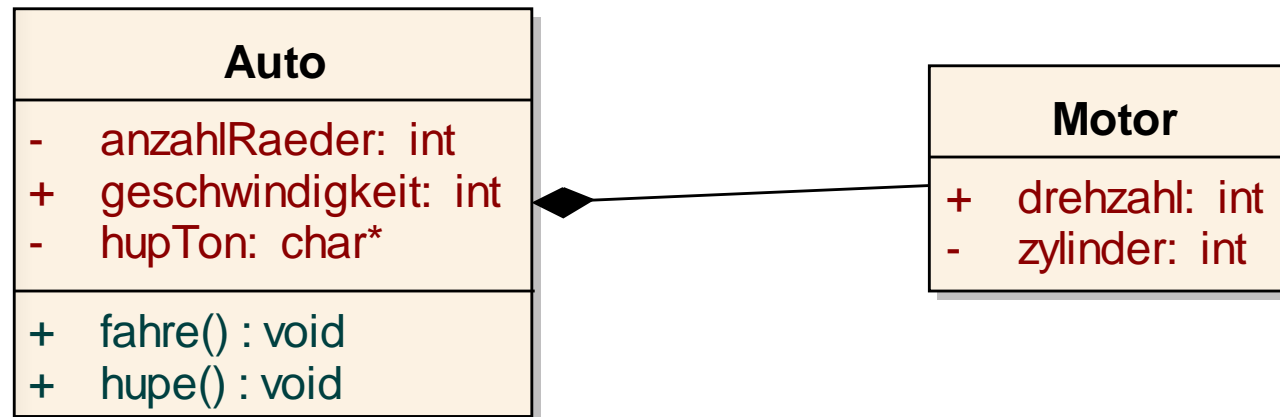
```
typedef struct
{
    int anzahlRaeder;
    char* hupTon;
    int geschwindigkeit;
}Auto;
```

Die Verbindung zwischen den Objekten wird erst zur Laufzeit des Programmes hergestellt.

```
Auto a;
Garage g;

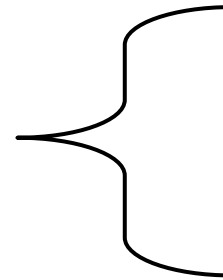
Auto_init(&a, 4, "tut");
Garage_init(&g);
Garage_einparken(&g, &a);
```

Die **Komposition** (Aggregation) ist eine „besteht aus“-Beziehung. Ein Objekt wird in ein anderes eingebettet.



Eine Komposition (Aggregation) wird realisiert, indem das einzubettende Objekt als **Member** in die Struktur des äußeren Objektes aufgenommen wird.

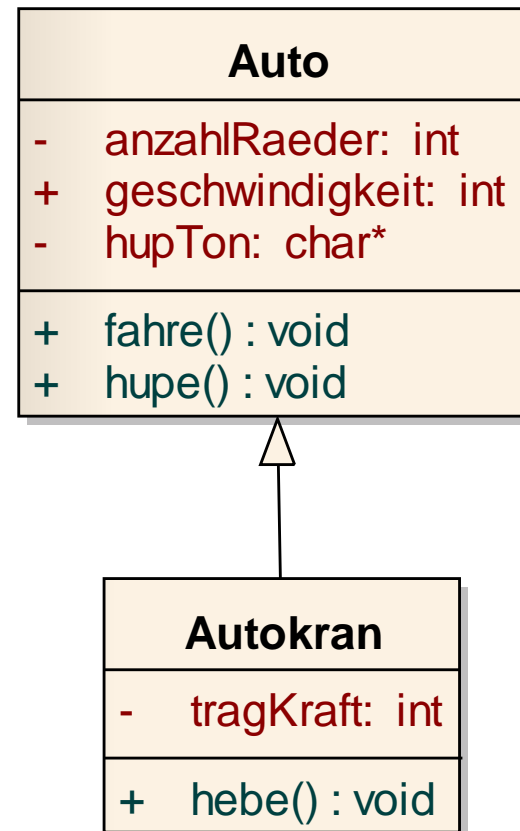
```
typedef struct
{
    int anzahlRaeder;
    char* hupTon;
    int geschwindigkeit;
    Motor motor;
}Auto;
```



```
typedef struct
{
    int zylinder;
    int drehzahl;
}Motor;
```

**Vererbung:**

Daten und Funktionen einer Basisklasse werden in einer abgeleiteten Klasse wiederverwendet und erweitert.



Basisklasse

abgeleitete Klasse

```
typedef struct
{
    int anzahlRaeder;
    char* hupTon;
    int geschwindigkeit;
    Motor motor;
}Auto;

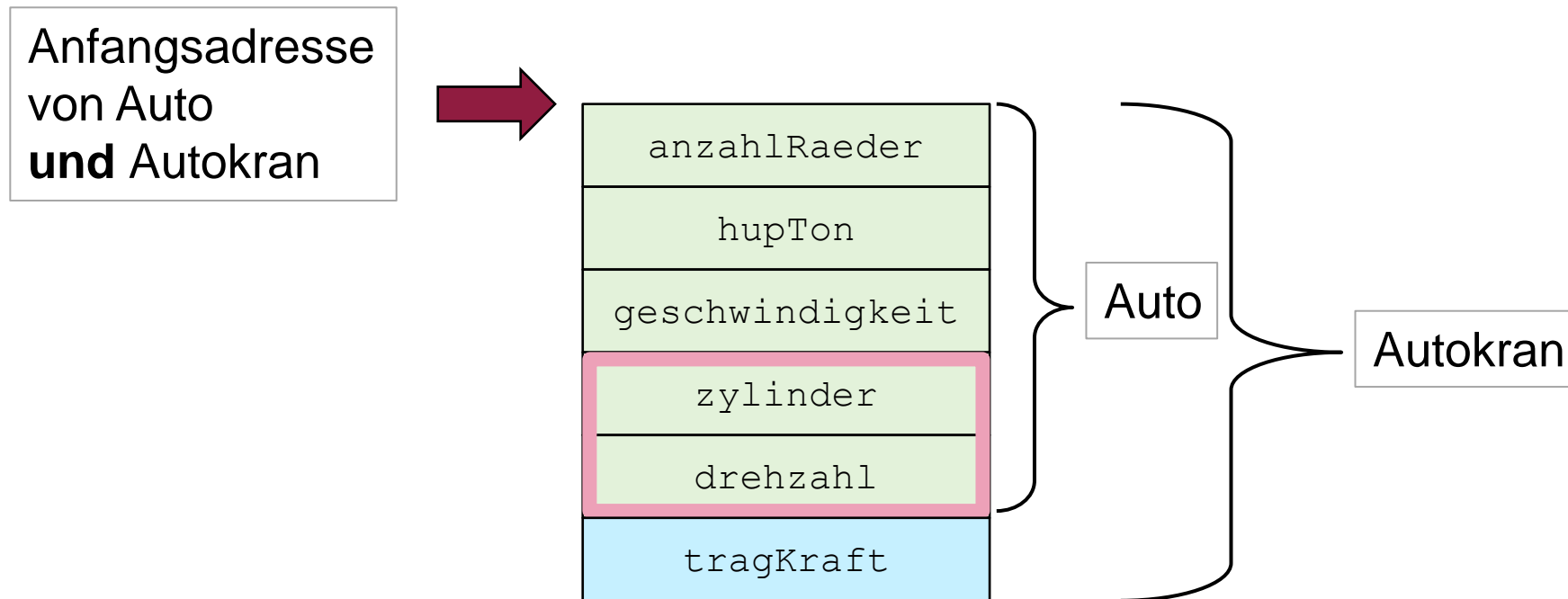
void Auto_init(Auto *self, int anzahlRaeder, char* hupTon);
void Auto_fahre(Auto *self);
void Auto_hupe(Auto *self, int anzahl);
void Auto_setGeschwindigkeit(Auto *self, int geschwindigkeit);
```

Die Vererbung in C ist ein Sonderfall der Aggregation (die Basisklasse ist das erste Element der Struktur).

```
typedef struct
{
    Auto;
    int tragKraft;
}Autokran;

void Autokran_init(Autokran *self, int tragKraft);
void Autokran_hebe(Autokran *self, int gewicht);
```

Die Struktur Autokran im Speicher:



Durch dieselbe Anfangsadresse können die Funktionen der Klasse Auto auch für Autokran-Objekte verwendet werden.

```
void test(void)
```

```
{
```

```
    Autokran kran;
```

```
    Autokran_init(&kran, 50);
```

```
    Auto_setGeschwindigkeit((Auto*)&kran, 20);
```

```
    Auto_hupe((Auto*)&kran, 3);
```

```
    Auto_setGeschwindigkeit((Auto*)&kran, 0);
```

```
    Autokran_hebe(&kran, 20);
```

```
    Autokran_hebe(&kran, 60);
```

```
}
```

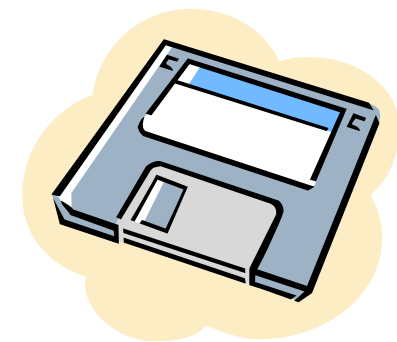
Ruft intern Auto\_init() auf

Funktionen der Basisklasse können problemlos benutzt werden.

Die Programmiersprache C kann mehr als auf den ersten Blick ersichtlich.

Setzen Sie moderne Methoden der SW-Entwicklung auch unter C ein.

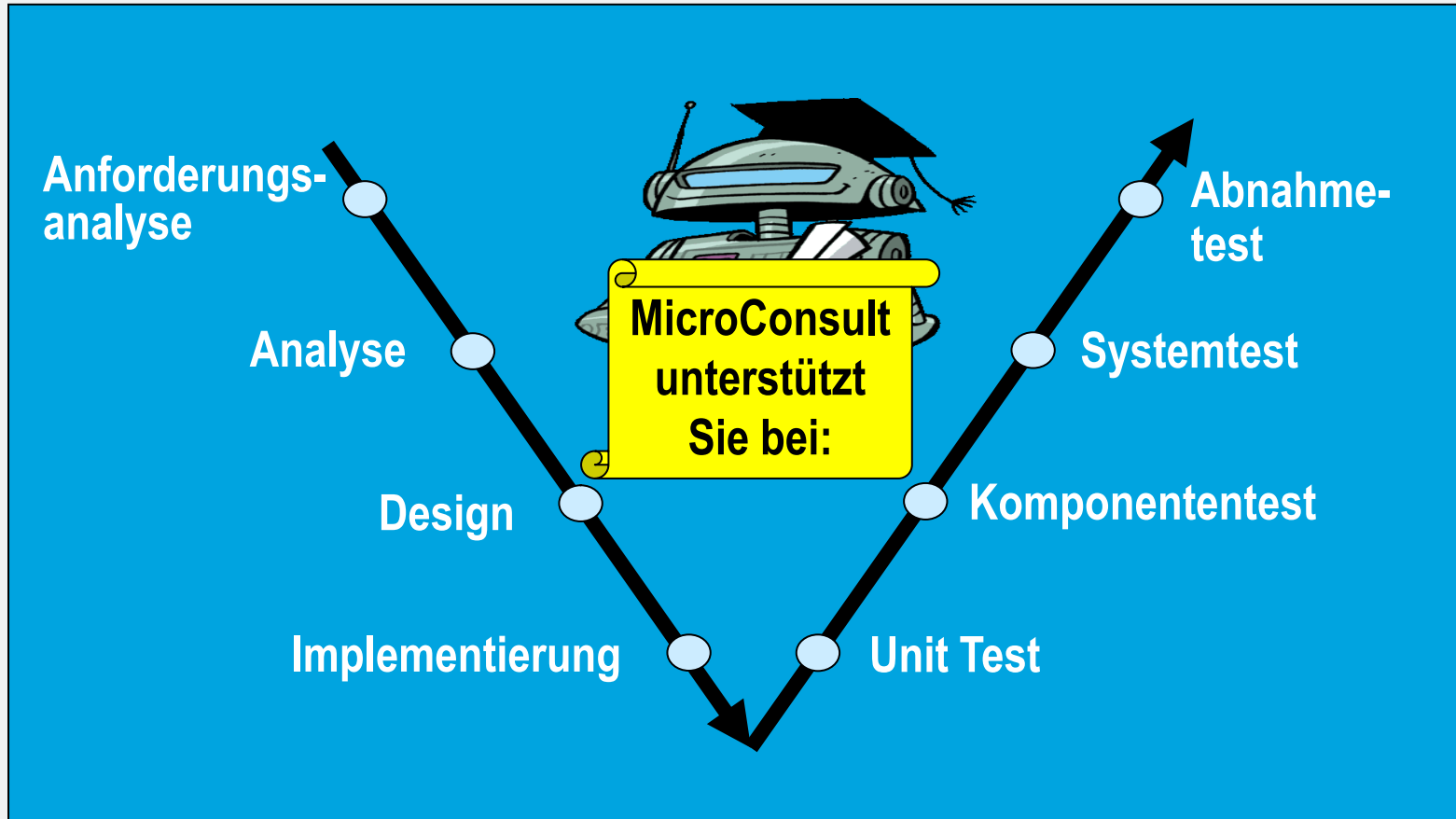
- sicherer Code
- wartbarer Code
- testbarer Code



## **TRAINING. COACHING. ENGINEERING.**

- UML Notation
- ANSI-C
- Embedded C
- ANSI-C++
- Embedded C++
- Modellbasierte Softwareentwicklung
- Softwareanalyse und Softwaredesign

Beratung, Training, Workshops, Coaching, Projektarbeit



HW-/SW-Technologien, Tools, Methoden, Prozess, Team