Klassen & Objekte

Komplexität & Abstraktion

In der Softwareprogrammierung ist es oft der Fall, dass man riesige Projekte mit Millionen von Zeilen Quellcode schreibt. Um diese äusserst komplexen Gebilde überhaupt noch überblicken zu können, wurde schon früh die Objektorientierung vorgeschlagen. Die Objektorientierung abstrahiert kleine Teilaufgaben einer komplexen, grossen Aufgabe. Dadurch lässt sich ein Programm modular aufbauen. Eine erste Implementation einer objektorientierten Sprache war Simula, welches von Ole-Johan Dahl und Kristen Nygaard zwischen 1962 und 1967 in Oslo entwickelt wurde. Anfangs war Simula lediglich als Ereignissimulationswerkzeug gedacht, aber allmählich entdeckte man, dass die Objektorientierung einiges Potential hatte. Simula wurde aber nie so erfolgreich, weil die Abstraktion der Objektorientierung ziemlich aufwändig in der berechnung ist – das heisst, in objektorientierten Sprachen geschriebene Programme kompilieren zu langsameren Programmen. Heute spielt diese Geschwindigkeitseinbusse meistens keine Rolle mehr und die wirtschaftlichen Vorteile sprechen für die Objektorientierung.

Aber vielleicht gibt es da immer noch eine bessere Möglichkeit, und wir wollen hoffen, dass ihr Informatiker der Zukunft diese einmal entdeckt; für eine schönere, bessere Welt.

Was ist Objektorientierung?

Bei der Objektorientierung werden Teilprobleme gelöst und passend verknüpft, um das ganze Problem lösen zu können. Diese Teilprobleme sind in Klassen aufgeteilt und für jede Klasse müssen wir eine Lösung finden:

herkömmliche Programme

objektorientierte Programme

Wie wir hier ganz deutlich sehen, sind herkömmliche Programme viel furchteinflössender als objektorientierte Programme. Nur steckt auch hier der Teufel im Detail: spielt ein kleines Monster nicht brav korrekt mit, wird das Programm fehlerhaft. Da aber diese kleine Monster weniger gross sind, sind sie auch nicht so gefährlich.

Die Monster in Eiffel

Die kleinen Teilprobleme werden in Eiffel als Klassen definiert. In jeder Klasse hat es verschiedene, aber trotzdem gleichartige Objekte. Eine Klasse definiert eine Menge von möglichen Objekten mit einem fest vorgegebenen Verhalten und einem streng definierten Zustandsraum.

Dies tönt vielleicht ein bisschen theoretisch und abstrakt, ist aber ganz einfach. Hier ein Beispiel mit Autos und Motoren (dieses typische Männerbeispiel liesse sich auch durch ein neutraleres Beispiel ersetzen,

aber ich möchte hier plakativ und einfach das Konzept der Objektorientierung erklären).

Autos & Motoren

Wenn wir ein Fahrzeug definieren müssten, sähe das in etwa so aus:

> *Ein Fahrzeug besteht aus einer Karrosserie.*

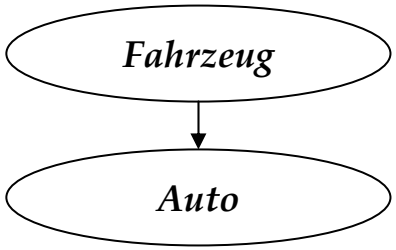
Wenn wir ein Autodefinieren definieren müssten, würden wir wahrscheinlich etwas ähnliche wie folgt sagen:

> *Autos bestehen grundsätzlich aus vier Rädern, einem Motor, den man starten kann und einer Karrosserie.*

oder:

> *Autos sind Fahrzeuge und haben einen Motor.*

Wie man feststellt, hängen die beiden Begriffe von einander ab:



- mehrere Räder und
- eine Karrosserie
- und ein Motor
- und Anzahl Räder ist 4

Diese Vererbung sieht in Eiffel folgendermassen aus:

```
class FAHRZEUG
feature
        eine_Karrosserie : KARROSSERIE
        -- und hier stünde noch einiges mehr...
end

class AUTO inherits FAHRZEUG
feature
        ein_Motor : MOTOR
        Motor_starten is
                do
                        -- hier stünde, wie der
                        -- Motor gestartet wird
                end
        links_vorne: RAD
        rechts_vorne: RAD
        links_hinten: RAD
        rechts_hinten: RAD
        -- hier wären auch noch einige Zeilen...
end
```

Die Klasse ***FAHRZEUG*** definiert, wie im Allgemeinen tatsächliche Exemplare von Fahrzeugen zur Laufzeit auszusehen haben. Diese Exemplare werden in der Informatik Objekte genannt.

Die Klasse definiert also

- mögliche Zustände von Objekten dieser Klasse, wie zum Beispiel Features wie ***Nummernschildkennzeichen***, ***gefahrene_Kilometer***
- Verhalten aller Objekte dieser Klasse, wie zum Beispiel Features wie ***Hupen***, ***Bremsen***, ***Kilometerstand_zurücksetzen***

Objekte der gleichen Klasse haben also das exakt gleiche Verhalten und die gleiche Auswahl an Zuständen bzw. bieten die identischen Features an.

Exemplare kreieren

Wenn man nun eine solche Klasse als Vorlage für zukünftige Objekte hat, möchte man sie dann vielleicht während der Laufzeit erstellen. Ein generelles Fahrzeug erstellten wir folgendermassen:

```
class ROOT_CLASS creation make
feature
        ein_Fahrzeug : FAHRZEUG    ❸
    ❶   make is
```

```
        -- unser Programm startet hier
        do
                create ein_Fahrzeug   ❷
        end
end
```

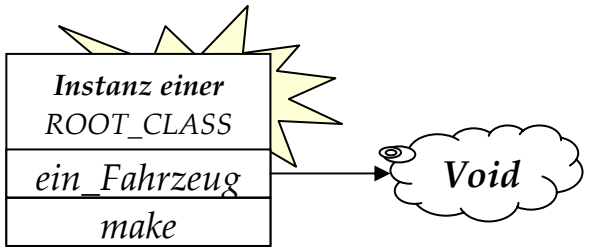
Die ***ROOT_CLASS*** ist jene Klasse, die beim Programmstart automatisch ausgeführt wird. Das heisst also, dass das Programm als erstes das Feature ***make*** (1) ausführt und damit in der Zeile (2) gestartet wird. Diese Zeile sieht so aus:

```
create ein_Fahrzeug
```

Mit dem blauen Wort ***create*** sagen wir dem Computer, dass das Feature ***ein_Fahrzeug*** instanziert werden soll – Instanzieren bedeutet, dass man eine Entity (ein Identifier mit einer Referenz auf ein Laufzeitobjekt) auf ein neues Laufzeitobjekt zeigen lässt.

Alle Entities wie zum Beispiel ***ein_Fahrzeug*** zeigen zu Beginn auf ein universales, leeres Ding, dem Nichts in Eiffel, und das nennt sich ***Void***. Also spielt sich bei Programmstart folgendes ab:

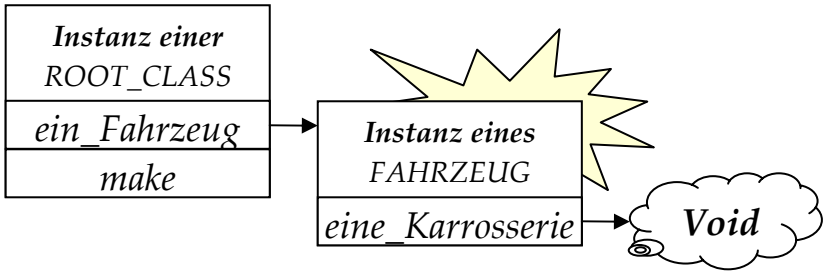
1. ein Objekt der ***ROOT_CLASS*** wird erstellt



2. das Command ***make*** wird aufgerufen und abgearbeitet

```
do
        create ein_Fahrzeug
end
```

3. ein Objekt der Klasse ***FAHRZEUG*** wird erstellt, weil die Entity ***ein_Fahrzeug*** vom Typ ***FAHRZEUG*** ist



4. die Entity ein_Fahrzeug zeigt jetzt auf die neu kreierte Instanz

Konstruktoren

Das Feature make macht also so etwas wie eine Initialisierung des Objektes der Klasse ***ROOT_CLASS***, damit keine seiner Entities mehr auf ein ***Void*** zeigen. Damit wir dies bereits mit dem create machen können bzw. müssen, gibt es die Möglichkeit, Konstruktoren zu definieren:

```
class FAHRZEUG
❶ create bauen
feature
        eine_Karrosserie : KARROSSERIE
    ❷ bauen is
                -- schauen, dass Karrosserie da ist
            do
                create eine_Karrosserie
            end
end
```

Jetzt ist das zuvor beschriebene ***create ein_Fahrzeug*** nicht mehr erlaubt. Mit der Zeile (1) definieren wir, dass Objekte der Klasse ***FAHRZEUG*** immer mit dem Feature ***bauen*** aufgerufen werden müssen. Neu wäre also korrekt:

```
create ein_Fahrzeug.bauen
```

Damit wird eine Instanz der Klasse ***FAHRZEUG*** erstellt, und im gleichen Zuge das Feature ***bauen*** aufgerufen.

Eine Klasse mit allen Schikanen

```
class
        AUTO
inherits
        FAHRZEUG
feature
        ein_Motor : MOTOR
        links_vorne: RAD
        rechts_vorne: RAD
        links_hinten: RAD
        rechts_hinten: RAD

        Motor_starten is
                require
                        not ein_Motor.laeuft
                do
                        ein_Motor.starter_betaetigen
                ensure
                        ein_Motor.laeuft
                end

        bauen is
                do
                        create ein_Motor
                        create links_vorne
                        create rechts_vorne
                        create links_hinten
                        create rechts_hinten
                end
invariant
        ein_Motor /= Void
end
```

Die Klasse ***Auto*** definiert Objekte, welche folgende Eigenschaften erfüllen:

- bestehen aus einem Feature ***eine_Karrosserie***, welche sie von der Mutterklasse ***FAHRZEUG*** geerbt haben
- bestehen aus den Features ***Motor_starten*** und bauen
- das Feature ***ein_Motor*** zeigt nie auf Void
- immer, wenn das Feature ***Motor_starten*** aufgerufen wird, muss ***ein_Motor.laeuft*** falsch sein
- immer, nachdem das Feature ***Motor_starten*** ausgeführt wurde, ist ***ein_Motor.laeuft*** wahr

Referenzen

Wikipedia Eintrag zur Objektorientierung – http://de.wikipedia.org/wiki/Objektorientierte_Programmierung

The History of Simula – http://java.sun.com/people/jag/SimulaHistory.html

Monsterbild – www.newf.com/_order-newfnet.php