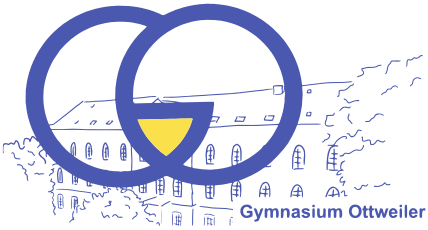
Kapitel 2

Objektorientiertes Modellieren und Programmieren



Skript Informatik · Grundkurs

Bisher bestand ein Programm aus Daten auf der einen und Unterprogrammen auf der anderen Seite. Das funktioniert, solange das Programm klein bleibt. Wächst es, wächst auch die Zahl der Stellen, an denen jemand an den Daten herumschreibt — und irgendwann weiß niemand mehr, wer den Kontostand zuletzt verändert hat.

Die Objektorientierung dreht die Anordnung um: Daten und die Unterprogramme, die auf ihnen arbeiten, werden zu einer Einheit verschweißt. Diese Einheit heißt **Objekt**, und von außen kommt man an ihre Daten nur noch heran, indem man das Objekt höflich darum bittet.

Das ist das größte Themenfeld des Grundkurses, und es ist weniger eine Programmiertechnik als eine Art, über Probleme nachzudenken.

Lernziele

Am Ende dieses Kapitels können Sie

- das Message-Passing-Modell erklären und zu einem vorgegebenen Sachzusammenhang ein solches Modell entwerfen,
- die Begriffe Klasse, Objekt, Instanz, Attribut und Methode sicher unterscheiden,
- Klassen mit Attributen einfacher und höherer Typen implementieren,
- Konstruktoren schreiben und Objekte durch Konstruktoraufruf erzeugen,
- Methoden mit und ohne Rückgabewert implementieren und dabei **Auftrag** und **Anfrage** unterscheiden,
- das Geheimnisprinzip erklären und begründen, warum der direkte Zugriff auf Attribute der Kernidee widerspricht,
- einzelne Klassen in UML-Notation darstellen und aus einem UML-Diagramm Quelltext schreiben,
- Vererbung einsetzen und erklären, wann sie angebracht ist.

2.1 Der Einstieg: ein Programm ohne Objekte

Versuch: Das Rollenspiel

Vier Personen bekommen je eine Karteikarte mit einer Zahl darauf — das ist ihr Kontostand. Sie dürfen auf ihre eigene Karte schauen und sie ändern, auf keine andere.

Die Klasse stellt nun Aufgaben: „Überweise 50 € von A nach C." „Wer hat am meisten?" „Hebe 200 € von B ab."

Einzige Regel: Es wird nur **gesprochen**. Niemand darf einem anderen in die Karte sehen oder hineinschreiben.

Notieren Sie, welche Sätze gefallen sind. Welche waren Fragen, welche Aufträge?

Auswertung

Was Sie gerade gespielt haben, ist das **Message-Passing-Modell** — das Grundmuster der Objektorientierung. Jede Person war ein Objekt: Sie hatte einen eigenen Zustand (die Zahl), sie kannte ihn als Einzige, und der einzige Weg zu ihr führte über Nachrichten.

Zwei Sorten Nachrichten sind dabei vorgekommen:

- **Anfragen** („Wie viel hast du?") — es kommt eine Antwort zurück, nichts ändert sich.
- **Aufträge** („Zahle 50 € ein.") — etwas ändert sich, eine Antwort braucht es nicht.

Und einmal wird jemand „Nein" gesagt haben: Wer 120 € hat, kann keine 200 € abheben. Diese Prüfung konnte nur die Person selbst vornehmen — sie allein kannte ihren Stand. Genau darin liegt der Gewinn.

Merke

Im Message-Passing-Modell ist der Austausch von Nachrichten zwischen Objekten der **einzige** Weg, auf dem Information fließt. Ein Objekt verwaltet seinen Zustand selbst und entscheidet selbst, welche Nachricht es wie beantwortet.

2.1.1 Wie es ohne Objekte aussähe

```
1 stand_a = 1200.0
2 stand_b = 120.0
3
4 # irgendwo im Programm, 400 Zeilen später
5 stand_b = stand_b - 200.0      # niemand hat geprüft
```

Das Programm läuft, und `stand_b` ist jetzt -80. Der Fehler ist nicht die eine Zeile — der Fehler ist, dass diese Zeile überhaupt geschrieben werden durfte. Die Prüfung steht woanders, vielleicht an fünf Stellen, vielleicht an vieren.

2.2 Klassen und Objekte

Definition: Klasse und Objekt

Eine **Klasse** ist der Bauplan: Sie legt fest, welche **Attribute** (Daten) und welche **Methoden** (Verhalten) die nach ihr gebauten Einheiten haben.

Ein **Objekt** (auch **Instanz**) ist eine nach diesem Bauplan erzeugte Einheit. Es belegt Speicher und trägt eigene Werte. Von einer Klasse kann es beliebig viele Objekte geben — gleich gebaut, verschieden gefüllt.

Der Vergleich, der trägt: Die Klasse ist das Formular, das Objekt das ausgefüllte Formular. Es gibt ein Formular und tausend Ausfüllungen.

Beispiel: Die Klasse Konto

Python

```python
class Konto:
    def __init__(self, inhaber):
        self.__inhaber = inhaber
        self.__stand = 0.0
```

Object Pascal

```
type
  TKonto = class
  private
    FInhaber: String;
    FStand: Real;
  public
    constructor Create(inhaber: String);
  end;
```

Zwei Konventionen, die Sie in fremdem Quelltext überall antreffen: In Object Pascal beginnen Klassennamen mit `T` (für *Type*) und Attributnamen mit `F` (für *Field*). Beides ist Vereinbarung, nicht Zwang — aber halten Sie sich daran, sobald jemand anderes Ihren Quelltext liest.

2.2.1 Der Konstruktor

Ein Objekt entsteht nicht dadurch, dass man eine Variable dafür deklariert. Es muss ausdrücklich erzeugt werden — durch den **Konstruktor**.

Definition: Konstruktor

Der **Konstruktor** ist die Methode, die ein neues Objekt erzeugt und seine Attribute mit Startwerten belegt. In Python heißt er `__init__`, in Object Pascal üblicherweise `Create`.

```
konto1 = Konto("A. Lovelace")
konto2 = Konto("C. Babbage")
```

```
constructor TKonto.Create(inhaber: String);
begin
  inherited Create;
  FInhaber := inhaber;
  FStand := 0.0;
end;

{ Aufruf: }
var konto1: TKonto;
begin
  konto1 := TKonto.Create('A. Lovelace');
  ...
  konto1.Free;        { Speicher wieder freigeben }
end;
```

Stolperstein

In Object Pascal ist eine Objektvariable ein **Zeiger**. Nach `var konto1: TKonto;` allein existiert noch kein Konto — `konto1` hat den Wert `nil`. Erst `TKonto.Create` legt das Objekt an. Und was erzeugt wurde, muss am Ende mit `Free` wieder freigegeben werden. Python räumt selbst auf, Pascal nicht.

Stolperstein

`konto2 := konto1;` kopiert **kein** Objekt. Danach zeigen beide Namen auf dasselbe Objekt, und das zweite ist unerreichbar geworden — sein Speicher bleibt belegt, ohne dass noch jemand hinkommt. Man nennt das ein **Speicherleck**.

Arbeitsauftrag 1

1. Erzeugen Sie drei Konten mit verschiedenen Inhabern und erklären Sie in einem Satz, was im Speicher geschieht.
2. Zeichnen Sie auf, was nach `konto2 := konto1;` im Speicher steht — vorher und nachher.

2.3 Methoden: Aufträge und Anfragen

Jetzt bekommt das Konto Verhalten. Die Unterscheidung aus Abschnitt 1.8 kehrt hier unverändert wieder.

Merke

Eine Methode **ohne** Rückgabewert ist ein **Auftrag** an das Objekt — es soll etwas tun. Eine Methode **mit** Rückgabewert ist eine **Anfrage** an das Objekt — es soll etwas mitteilen.

Python

```python
class Konto:
    def __init__(self, inhaber):
        self.__inhaber = inhaber
        self.__stand = 0.0

    def einzahlen(self, betrag):            # Auftrag
        if betrag > 0:
            self.__stand = self.__stand + betrag

    def abheben(self, betrag):              # Anfrage: hat es geklappt?
        if betrag <= 0 or betrag > self.__stand:
            return False
        self.__stand = self.__stand - betrag
        return True

    def stand(self):                        # Anfrage
        return self.__stand

    def inhaber(self):                      # Anfrage
        return self.__inhaber
```

Object Pascal

```
type
  TKonto = class
  private
    FInhaber: String;
    FStand: Real;
  public
    constructor Create(inhaber: String);
    procedure Einzahlen(betrag: Real);
    function Abheben(betrag: Real): Boolean; virtual;
    function Stand: Real;
    function Inhaber: String;
  end;

procedure TKonto.Einzahlen(betrag: Real);
begin
  if betrag > 0 then
    FStand := FStand + betrag;
end;

function TKonto.Abheben(betrag: Real): Boolean;
begin
  if (betrag <= 0) or (betrag > FStand) then
```

```
23      Result := False
24    else
25    begin
26      FStand := FStand - betrag;
27      Result := True;
28    end;
29  end;
30
31  function TKonto.Stand: Real;
32  begin
33    Result := FStand;
34  end;
```

`abheben` ist der interessante Fall: Es *tut* etwas und *meldet* zugleich, ob es geklappt hat. Diese Mischform ist erlaubt und üblich — der Rückgabewert sagt nichts über den Zustand aus, sondern nur über den Erfolg des Auftrags.

Beispiel: Die Prüfung sitzt jetzt an genau einer Stelle

```
1 konto = Konto("A. Lovelace")
2 konto.einzahlen(120.0)
3
4 if konto.abheben(200.0):
5     print("ausgezahlt")
6 else:
7     print("Deckung reicht nicht:", konto.stand(), "€")
```

Ausgabe:

```
  Deckung reicht nicht: 120.0 €
```

Egal, von wie vielen Stellen im Programm abgehoben wird — geprüft wird immer hier, in der Methode `abheben`. Es gibt keine zweite Möglichkeit, an `__stand` heranzukommen.

2.4 Datenkapselung und das Geheimnisprinzip

In beiden Fassungen stehen die Attribute geschützt: in Pascal hinter `private`, in Python hinter den beiden Unterstrichen von `__stand`.

Definition: Geheimnisprinzip

Nach dem **Geheimnisprinzip** verbirgt ein Objekt seine Daten vor der Außenwelt. Zugriff gibt es ausschließlich über Methoden, die das Objekt selbst anbietet. Man nennt das **Datenkapselung**.

Zwei Gründe, und der zweite ist der wichtigere:

Erstens die Prüfung. `konto.stand = -80` ist nicht möglich. Wer etwas am Stand ändern will, muss durch `einzahlen` oder `abheben`, und dort wird geprüft.

Zweitens die Freiheit, es später anders zu machen. Solange niemand von außen an `__stand` heranreicht, dürfen Sie das Attribut jederzeit umbauen — in Cent statt Euro rechnen, es aus einer Datei nachladen, es aus einer Liste von Buchungen berechnen. Kein fremder Quelltext bricht dabei. Hätten hundert Stellen direkt auf `__stand` zugegriffen, müssten Sie hundert Stellen ändern.

Merke

Ein direkter Zugriff auf ein Attribut von außen widerspricht der Kernidee des Message-Passing-Modells. Er verhindert jede spätere Umorganisation des Objektinneren.

Stolperstein

Pythons `__stand` ist kein echter Schutz — wer will, kommt mit `konto._Konto__stand` heran. Das ist Absicht: Python verlässt sich auf Übereinkunft statt auf Verbote. Die Übereinkunft gilt trotzdem. Was mit Unterstrichen beginnt, fasst man nicht an.

2.4.1 Lesen erlauben, Schreiben prüfen

Oft soll ein Attribut von außen *lesbar*, aber nicht frei *beschreibbar* sein. Object Pascal hat dafür die **Eigenschaft** (`property`):

```
type
  TKonto = class
  private
    FInhaber: String;
    FStand: Real;
    procedure SetzeInhaber(wert: String);
  public
    property Inhaber: String read FInhaber write SetzeInhaber;
    property Stand: Real read FStand;          { nur lesen }
  end;

procedure TKonto.SetzeInhaber(wert: String);
begin
  if wert <> '' then
    FInhaber := wert;
end;
```

Von außen sieht `Inhaber` aus wie ein ganz normales Attribut — tatsächlich läuft beim Schreiben die Prüfung in `SetzeInhaber`. `Stand` hat kein `write` und ist damit nur lesbar.

Python erreicht dasselbe mit `@property`:

```
1 class Konto:
2     @property
3     def stand(self):
4         return self.__stand
```

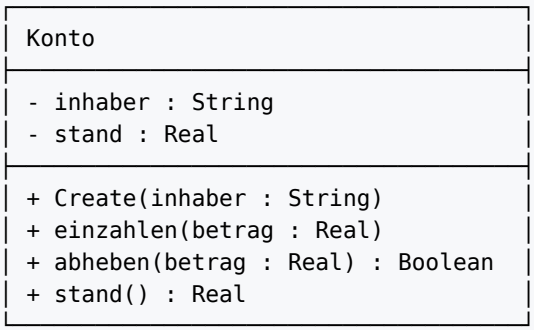
Arbeitsauftrag 2

3. Ergänzen Sie die Klasse `Konto` um eine Methode `ueberweisen(ziel, betrag)`, die Geld auf ein anderes Konto überweist. Benutzen Sie dabei ausschließlich vorhandene Methoden — greifen Sie **nicht** auf `__stand` des Zielkontos zu.
4. Was soll `ueberweisen` zurückgeben? Begründen Sie Ihre Entscheidung mit den Begriffen Auftrag und Anfrage.
5. Begründen Sie an einem eigenen Beispiel, warum ein Attribut `alter` in einer Klasse `Person` besser nicht öffentlich beschreibbar ist.

Gymnasium Ottweiler

2.5 Die UML-Notation

Vor dem Programmieren steht das Modellieren. Dafür hat sich eine Zeichensprache durchgesetzt: die **Unified Modeling Language**, kurz UML. Eine Klasse ist darin ein Kasten aus drei Feldern.



Die Zeichen davor geben die Sichtbarkeit an:

Zeichen	Bedeutung	Pascal	Python
`-`	privat — nur innerhalb der Klasse	`private`	`__name`
`+`	öffentlich — für alle	`public`	`name`
`#`	geschützt — Klasse und ihre Nachkommen	`protected`	`_name`

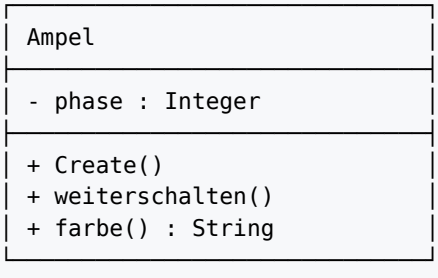
Merke

Ein Methodeneintrag **ohne** Typ hinter dem Doppelpunkt ist ein **Auftrag**, einer **mit** Typ eine **Anfrage**. Das UML-Diagramm zeigt die Unterscheidung also unmittelbar.

Und die gesunde Faustregel ist im Diagramm sofort sichtbar: Attribute tragen `-`, Methoden `+`.

Arbeitsauftrag 3

6. Zeichnen Sie das UML-Diagramm zu einer Klasse `Schueler` mit Name, Klassenstufe und einer Liste von Noten. Überlegen Sie zuerst, welche Nachrichten ein Schüler-Objekt beantworten können muss.
7. Schreiben Sie zu folgendem Diagramm den Quelltext in einer Sprache Ihrer Wahl:

   `phase` läuft von 0 bis 3 und bedeutet rot, rot-gelb, grün, gelb.

2.6 Objekte, die andere Objekte kennen

Ein Attribut muss keine Zahl sein. Es kann selbst ein Objekt sein — oder eine ganze Liste davon. Erst dadurch entstehen Modelle, die mehr als ein Ding beschreiben.

Beispiel: Eine Bank hält ihre Konten

```
class Bank:
    def __init__(self, name):
        self.__name = name
        self.__konten = []                      # Liste von Konto-Objekten

    def aufnehmen(self, konto):                 # Auftrag: nimmt ein fertiges
    Konto auf
        self.__konten.append(konto)
        return konto

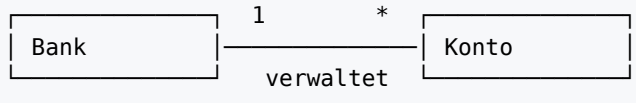
    def eroeffnen(self, inhaber):               # Anfrage: liefert das neue Konto
        return self.aufnehmen(Konto(inhaber))

    def gesamtvermoegen(self):                  # Anfrage
        summe = 0.0
        for konto in self.__konten:
            summe = summe + konto.stand()       # Nachricht an jedes Konto!
        return summe

    def anzahl(self):
        return len(self.__konten)
```

Beachten Sie die Zeile mit `konto.stand()`. Die Bank weiß nicht, *wie* ein Konto seinen Stand verwaltet — sie fragt danach. Das ist Message-Passing in zwei Zeilen: Ein Objekt erfüllt seine Aufgabe, indem es anderen Objekten Nachrichten schickt.

Im UML-Diagramm wird diese Beziehung als Linie zwischen den Klassen gezeichnet, mit der Zahl der beteiligten Objekte an den Enden:



Gelesen: Eine Bank verwaltet beliebig viele (*) Konten; jedes Konto gehört zu genau einer Bank.

Arbeitsauftrag 4

8. Erweitern Sie `Bank` um eine Anfrage `reichster()`, die das Konto mit dem höchsten Stand zurückgibt. Was soll bei einer Bank ohne Konten geschehen?
9. Modellieren Sie eine Schulbibliothek: Welche Klassen brauchen Sie, welche Attribute, welche Nachrichten gehen zwischen den Objekten hin und her? Zeichnen Sie das UML-Diagramm, bevor Sie eine Zeile Quelltext schreiben.

2.7 Vererbung

LK

LK

Vererbung und Polymorphie stehen im Themenfeld *Objektorientierung* nur im Lehrplan des Leistungskurses. Der Grundkurs endet bei Klassen, Objekten, Attributen, Konstruktoren und Methoden samt UML-Darstellung einzelner Klassen.

Zwei Klassen ähneln einander oft stark. Ein Girokonto und ein Sparkonto haben beide Inhaber und Stand; nur das Abheben läuft verschieden. Alles doppelt zu schreiben wäre Arbeit und würde beim nächsten Fehler zweimal korrigiert werden müssen.

Definition: Vererbung

Eine Klasse kann von einer anderen **erben**. Die **Unterklasse** übernimmt alle Attribute und Methoden der **Oberklasse** und darf eigene hinzufügen oder geerbte überschreiben.

Die Probe aufs Exempel ist immer derselbe Satz: „*Ein Sparkonto **ist ein** Konto.*" Stimmt der Satz, passt die Vererbung. Stimmt er nicht, nehmen Sie ein Attribut. (Ein Konto *ist* kein Inhaber — es *hat* einen.)

Python

```python
class Sparkonto(Konto):
    def __init__(self, inhaber, zinssatz):
        super().__init__(inhaber)               # Konstruktor der Oberklasse
        self.__zinssatz = zinssatz

    def verzinsen(self):                        # neu
        self.einzahlen(self.stand() * self.__zinssatz / 100)

    def abheben(self, betrag):                  # überschrieben
        if betrag > 500:
            return False                        # Sparkonto: höchstens 500 €
        return super().abheben(betrag)
```

Object Pascal

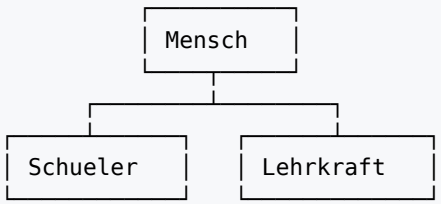
```
type
  TSparkonto = class(TKonto)
  private
    FZinssatz: Real;
  public
    constructor Create(inhaber: String; zinssatz: Real);
    procedure Verzinsen;
    function Abheben(betrag: Real): Boolean; override;
  end;

constructor TSparkonto.Create(inhaber: String; zinssatz: Real);
begin
  inherited Create(inhaber);
  FZinssatz := zinssatz;
end;

function TSparkonto.Abheben(betrag: Real): Boolean;
begin
  if betrag > 500 then
    Result := False
  else
    Result := inherited Abheben(betrag);
end;
```

`inherited` beziehungsweise `super()` ruft die Fassung der Oberklasse auf. So muss die Prüfung auf ausreichende Deckung kein zweites Mal geschrieben werden — das Sparkonto fügt nur seine eigene Zusatzregel hinzu und reicht dann weiter.

Beispiel: Eine Vererbungshierarchie



`Mensch` trägt Vorname, Nachname, Geburtsdatum. `Schueler` ergänzt Klassenstufe, `Lehrkraft` ergänzt Fächer. Was beide gemeinsam haben, steht genau einmal da.

2.7.1 Polymorphie

Weil ein Sparkonto ein Konto *ist*, darf es überall stehen, wo ein Konto erwartet wird. Die Bank aus Abschnitt 2.6 funktioniert unverändert auch mit Sparkonten in ihrer Liste.

```
bank = Bank("Sparkasse")
giro = bank.eroeffnen("A. Lovelace")        # ein gewöhnliches Konto
spar = Sparkonto("C. Babbage", 1.5)
bank.aufnehmen(spar)                        # ein Sparkonto in derselben Liste

giro.einzahlen(1000)
spar.einzahlen(1000)
print(giro.abheben(600))                    # True  – Konto.abheben
print(spar.abheben(600))                    # False – Sparkonto.abheben
print(bank.gesamtvermoegen())               # 1400.0
```

Zweimal dieselbe Nachricht `abheben(600)`, zweimal ein anderes Ergebnis — und die Bank hat dafür keine einzige Zeile geändert. Welche der beiden Fassungen läuft, entscheidet sich erst zur Laufzeit, und zwar am Empfänger der Nachricht. Das nennt man **Polymorphie** („Vielgestaltigkeit").

Stolperstein

In Python geschieht das von allein. In Object Pascal **nicht**: Eine Methode wird nur dann zur Laufzeit ausgewählt, wenn sie in der Oberklasse als `virtual` und in der Unterklasse als `override` gekennzeichnet ist. Deshalb steht oben in `TKonto` `function Abheben(...): Boolean; virtual;` und in `TSparkonto ... override;`. Fehlt das `virtual`, lehnt der Übersetzer das `override` ab; schreibt man stattdessen bloß eine gleichnamige Methode ohne beides, übersetzt alles anstandslos — und es läuft trotzdem immer die Fassung der Oberklasse, sobald das Objekt in einer Variablen vom Typ `TKonto` steckt. Das ist einer der am schwersten zu findenden Fehler in objektorientiertem Pascal.

Merke

Polymorphie ist der eigentliche Gewinn der Vererbung. Der aufrufende Quelltext muss nicht wissen, welche Art von Konto vor ihm steht. Er schickt die Nachricht, das Objekt weiß selbst, was zu tun ist.

Arbeitsauftrag 5

10. Ergänzen Sie `Girokonto` mit einem Dispo-Rahmen von 1000 €, bis zu dem der Stand negativ werden darf.
11. Fügen Sie der Bank ein Girokonto und ein Sparkonto hinzu und rufen Sie auf beiden `abheben(600)` auf. Erklären Sie die unterschiedlichen Ergebnisse.
12. Prüfen Sie mit dem *ist-ein*-Satz, ob Vererbung passt:

    a) Quadrat — Rechteck, b) Auto — Motor, c) Lehrkraft — Mensch, d) Liste — Baum.

2.8 Zusammenfassung

Begriff	Kurz gesagt
Klasse	der Bauplan
Objekt / Instanz	eine nach dem Bauplan erzeugte Einheit mit eigenen Werten
Attribut	Datenfeld eines Objekts, beschreibt seinen Zustand
Methode	Unterprogramm einer Klasse, beschreibt ihr Verhalten
Konstruktor	erzeugt ein Objekt und setzt die Startwerte
Auftrag	Methode ohne Rückgabewert — das Objekt soll etwas tun
Anfrage	Methode mit Rückgabewert — das Objekt soll etwas mitteilen
Datenkapselung	Attribute privat, Zugriff nur über Methoden
Vererbung	Unterklasse übernimmt alles von der Oberklasse
Polymorphie	dieselbe Nachricht, je nach Empfänger anderes Verhalten

Selbstcheck

Können Sie an Ihrem Bibliotheksmodell aus Aufgabe 9 zeigen, welche Methode ein Auftrag und welche eine Anfrage ist? Und können Sie ohne nachzusehen begründen, warum ein öffentliches Attribut spätere Änderungen am Objektinneren verhindert?

Gymnasium Ottweiler