

# Einführung in die Programmieretechnik

## Objektorientierte Programmierung

# Programmieren im Großen

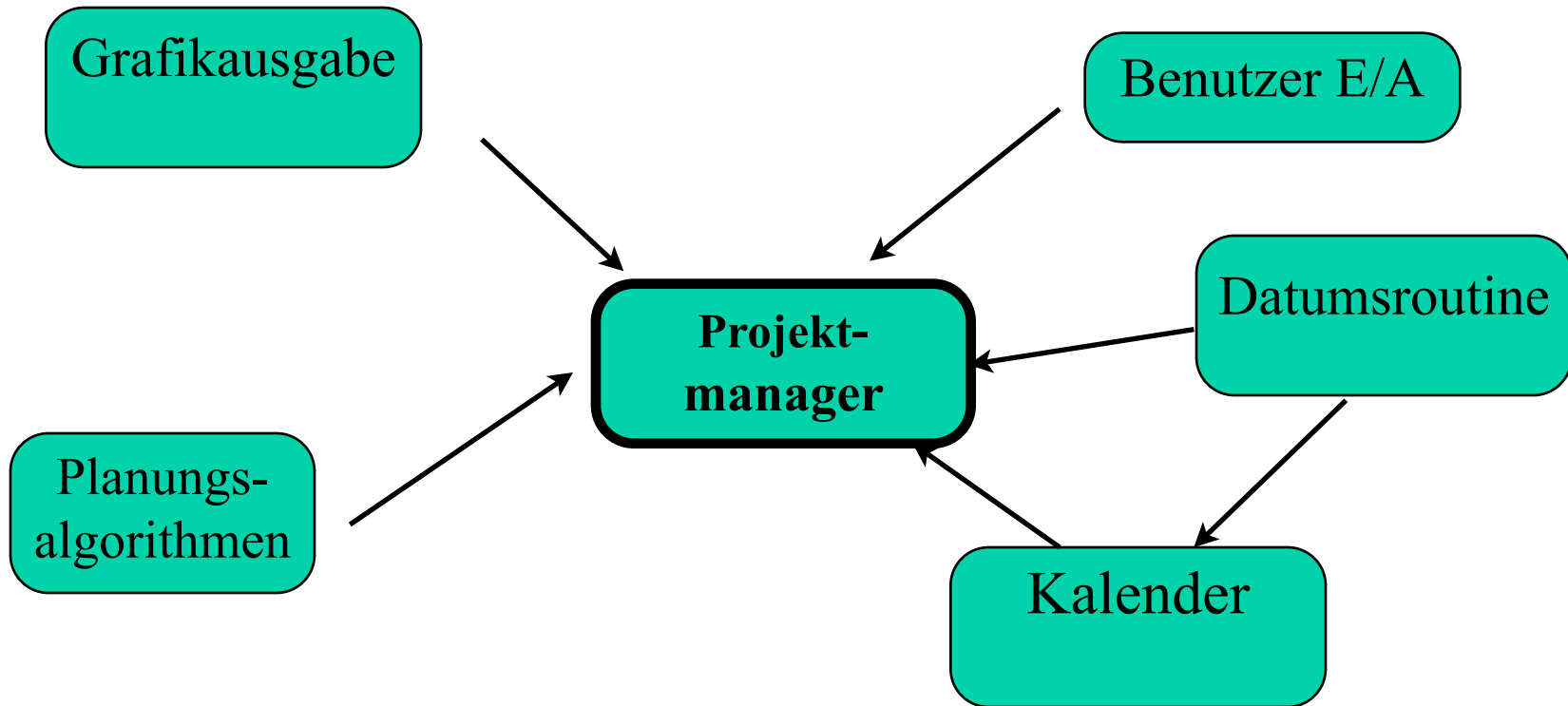
- **Strukturierte Programmierung: Blockkonzept (ab 1960)**
  - Algorithmen können in Teilalgorithmen zerlegt werden
- **Große Systeme: auch strukturierte Programme werden unübersichtlich**
  - Entwicklung durch einzelnen Autor nicht mehr möglich
  - Lebenszeit des Softwaresystems u.U. länger als ursprünglicher Autor am System arbeitet
- **Lösungsansatz: Modulare Programmierung (ab 1970)**
  - Arbeitsteilung: Entwickler sind für verschiedene Module zuständig
  - Entkopplung von Entwicklungszyklen: einzelne Module können separat fertiggestellt und revidiert werden
- **Lösungsansatz: Objektorientierte Programmierung (ab 1980)**
  - ursprünglich getrieben von interaktiven graphischen Systemen (Maus)

# Modulare Programmierung

- Zerlegung des Gesamtsystems in Teilsysteme
  - üblich: Funktionsblöcke
- Spezifikation der einzelnen Module
  - Definition der Schnittstelle: Welche Funktionen/Prozeduren werden angeboten? Welche Datentypen sind Parameter und Ergebnis
  - Beschreibung der Semantik jedes Moduls
- Integration der Module
  - DeRemer, Kron: Programming In the Large versus Programming In the Small, 1975
    - Module in verschiedenen Programmiersprachen entwickelt
    - “module interconnection language”: MIL 75
  - “glue languages”: Skriptsprachen, z.B. VisualBasic, Python...
  - heute: Sprachen unterstützen sowohl Moduldefinition als auch Modulintegration
- Komponentenbasierte Programmierung

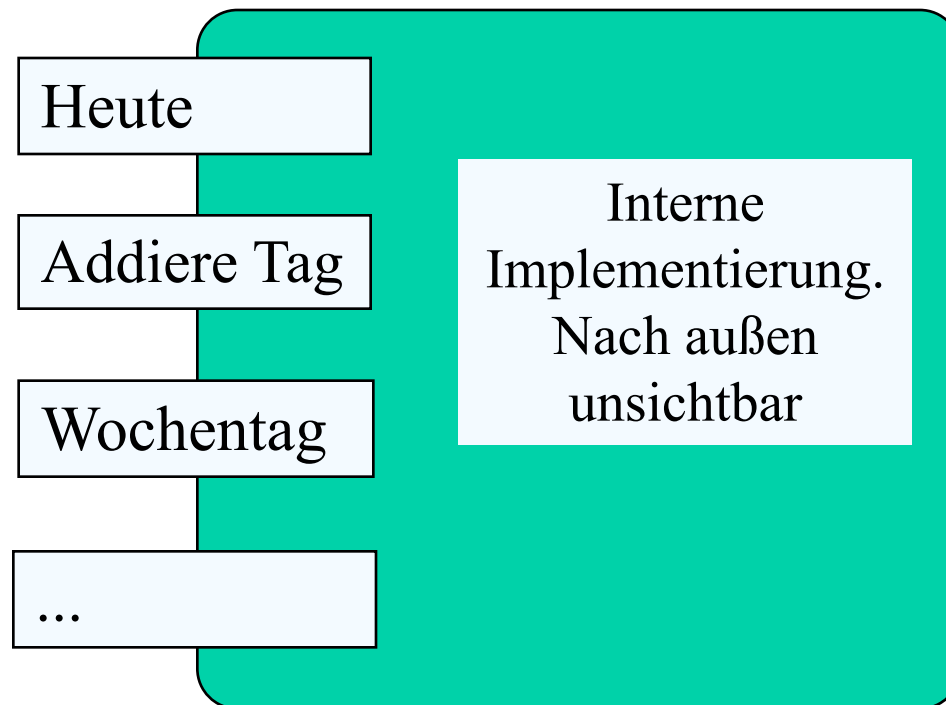
# Modulare Programmierung (2)

- Beispiel: ein Projektmanager



# Modulare Programmierung (3)

- Schnittstelle eines Kalendermoduls



# Module in Python

- Implementiert in C oder Python
- “Bibliothek”: i.d.R. bietet ein Modul alle Funktionen zu einem Thema
- Verwendung von Modulen: import

import\_stmt ::=

```
"import" module ["as" name] ( "," module ["as" name] )*  
| "from" module "import" identifier ["as" name]  
  ( "," identifier ["as" name] )*  
| "from" module "import" "(" identifier ["as" name]  
  ( "," identifier ["as" name] )* [","] ")"  
| "from" module "import" ""
```

module ::= (identifier ".")\* identifier

# Module in Python (2)

- Python-Module: `<modulname>.py`
  - import-Anweisung führt den Modulinhalt aus
- C-Module: eingebaut, oder plattform-spezifische Dateien
  - Windows: `<modulname>.pyd` / `<modulname>.dll`
  - Linux: `<modulname>.so`
- Suchpfad für Module: `sys.path`
  - beeinflusst von Umgebungsvariable `PYTHONPATH`
  - eingebaute Module: `sys.builtin_module_names`
- Dictionary der importierten Module: `sys.modules`
- Inhalt von Modulen: Variablen, Funktionen, Klassen
  - jeweils mit Dokumentation
- Moduldokumentation: Erstes String-Literal in der Pythondatei

# Module in Python (3)

- Pakete (*packages*): Zusammenfassungen von Modulen
  - Verzeichnis mit Spezialmodul `__init__.py`
- “import paket” lädt nur `__init__.py`
- “import paket.modul” lädt zunächst Paket, dann Modul
- import-Anweisung führt neue Namen ein:
  - import a: Variable a wird an Modul gebunden
  - import a.b: Variable a wird an Modul a gebunden; a.b wird importiert
  - from a import b: Modul a wird importiert; dann wird Wert von a.b in den aktuellen Namensraum eingeführt

```
import a.b
b = a.b
del a
```
  - from a import b as c: a.b wird unter dem Namen c importiert
  - from a import \*: Modul a wird importiert, dann werden alle Werte im Modul a in den aktuellen Namensraum importiert



# Objektorientiertes Programmieren (OOP)

- *Objekte = Daten + Methoden*
  - genauer: Daten + Methoden + Identität
- Objekte sind nicht nur reine Datensätze, sondern enthalten auch (Verweise auf) Operationen für die Daten
- *Methoden* sind Unterprogramme, die für einen und innerhalb eines Datentyps definiert sind
  - Kapselung/Verkapselung (*encapsulation*): Zusammenfassung der Daten und Operationen zu einem abstrakteren Typ
- *Klasse*: Zusammenfassung von Datensatzstruktur und Methoden
  - Daten heißen *Felder* (fields) oder *Attribute* (*attributes*) der Klasse, oft auch *members*

# OOP (2)

- *Objekt*: konkrete Belegung der Felder
  - oft auch *Exemplar* (instance):
    - Ein Exemplar der Klasse `Button`
    - seltener: Ein Objekt der Klasse `Button`
  - fälschlich oft *Instanz*

In|s|tanz, die; -, -en <lat.> (zuständige Stelle bei Behörden oder Gerichten); In|s|tan|zen|weg (Dienstweg) [Quelle: Duden, 22. Auflage, 2000]

-

# Klassen

- doppelte Verwendung des Begriffs “Klasse”:
  1. Teil einer Klassifikation
  2. Programmkonstrukt zur Kapselung und Wiederverwendung von Datenstrukturen und Algorithmen
- Klassifikationen: Unterteilung einer Menge in Teilmengen
  - z.B. mittels Äquivalenzrelation in Äquivalenzklassen
  - z.B. mittels Taxonomie in Taxa (*sing.* Taxon, Gruppe)
    - Biologie: Rang einer Gruppe: Art, Gattung, Familie
  - synonym: Systematik
  - Einordnung eines Objekts in eine Klasse: Klassierung
    - jedes Objekt kann u.U. zu mehreren Klassen gehören, u.U. auch seine Klassenzugehörigkeit mit der Zeit ändern
- Klassen in der OOP: u.U. auch Klassifikation
  - aber: Programmstruktur steht im Vordergrund

# Datenkapselung

- Ziel: Verstecken der Datenstrukturen (*information hiding*)
  - Austausch der Repräsentation von Daten ohne Änderung der Methodensignaturen wird möglich
    - Beispiel: Punkte auf der Ebene entweder in kartesischen oder Polarkoordinaten repräsentierbar
- Ziel: Durchsetzung von Regeln für Daten unabhängig von Anwendungsprogramm (Protokolle, *protocols*)
  - z.B. Wahrung von Invarianten
    - “Der Kontostand muss immer oberhalb des Dispositionscredits sein.”
  - z.B. Durchsetzung der Buchführung (*logging*)
- Ziel: Gleichzeitige Verwendung unterschiedlicher Realisierungen eines Protokolls (Polymorphie, *polymorphism*)
  - z.B. GUI: verschiedene Klassen (`Button`, `Label`, `PictureBox`) implementieren alle eine Methode `.Show`)

# Datenkapselung (2)

- Beispiel: Invariante für Kontoklasse

```
class Konto:
```

```
    def __init__(self, dispo):
```

```
        self.stand = 0
```

```
        self.dispo = dispo
```

```
    def lesen(self):
```

```
        return self.stand
```

```
    def einzahlen(self, betrag):
```

```
        log("Einzahlung", betrag)
```

```
        self.stand += betrag
```

```
    def abheben(self, betrag):
```

```
        if self.stand-betrag < -self.dispo:
```

```
            raise Ueberziehung()
```

```
        log("Auszahlung", betrag)
```

```
        self.stand -= betrag
```

# Vererbung

- Erweiterung einer bestehenden Klasse um neue Eigenschaften (neue Daten, neue Methoden)
  - nicht durch Änderung der Klasse, sondern durch Definition einer neuen Klasse
  - neue Klasse: *erweiterte* oder *abgeleitete* oder *Unterklasse* (*subclass*)
  - bestehende Klasse: *Basis-* oder *Oberklasse* (*superclass*)
- Beispiel:
  - Basisklasse: Verzeichniseintrag (Attribute: name, besitzer, datum der letzten Änderung)
  - Ableitungen:
    - Datei (Attribut: Größe, Inhalt)
      - Methoden: read, write
    - Verzeichnis (Attribute: Liste von Verzeichniseinträgen)
      - Methoden: create\_file, make\_directory, read\_entries

# Interpretation von Vererbung

- Vererbung im juristischen Sinne:
  - Eigentum geht in Besitz des Erben über
- Vererbung im biologischen Sinne:
  - Erbgut wird als Kopie in neues Leben übergeben
  - Eltern besitzen weiterhin das Erbgut
- OOP-Begriff angelehnt an biologischen Begriff

# Polymorphie

- von “ πολυμορφία ”: vielgestaltig
- Ziel: Austausch von Klassen als Parameter eines Algorithmus, ohne Änderung der Klasse
- Beispiel: Bestimmung der 2D-Objekte, die sich an einem bestimmten Punkt befinden

```
p = Point(10,7)
```

```
for o in objekte:
```

```
    if o.enthaelt(p):
```

```
        print o.als_text()
```



# Polymorphie (2)

- Geometrische Objekte: Bestimmt durch geschlossene Linie
  - Form der Linie abhängig von Objekt
  - allgemeine, objekt-unabhängiges Kalkül für geschlossene Linien schwer realisierbar
  - Betrachtung von “interessanten” Spezialfällen:
    - Kreise, achsenparallele Rechtecke
    - Verallgemeinerung: Ellipsen, Vielecke
    - Verallgemeinerung: Linienzüge auf Basis von Strecken, Kreisbögen, Splines
  - Thema der Computergrafik
    - hier nur Kreise, achsenparallele Rechtecke betrachtet

# Beispiel: 2-D-Objekte

- Definition der Basisklasse (Zwo\_D\_Objekt)
  - Annahme: Jedes Objekt kann einen Namen haben
  - Textausgabe: Objekt gibt Namen aus
  - Enthaltensein von Punkten: Im allgemeinen nicht beantwortbar
    - abstrakte Methode

```
class Zwo_D_Objekt:
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
    def als_text(self):
```

```
        return self.name
```

```
    def enthaelt(self, punkt):
```

```
        raise NotImplementedError(self.name + " hat keinen Algorithmus für  
        enthaelt definiert")
```

# Beispiel: Kreise

- Repräsentation durch Mittelpunkt und Radius
- Enthaltensein anhand von Kreisformel
  - Annahme: Randpunkte sind auch “enthalten”

```
class Kreis(Zwo_D_Objekt):
    def __init__(self, mittelpunkt, radius):
        Zwo_D_Objekt.__init__(self, "kreis")
        self.mittelpunkt = mittelpunkt
        self.radius = radius
    def als_text(self):
        return "kreis("+self.mittelpunkt.als_text()+", "+str(self.radius)+")"
    def enthaelt(self, punkt):
        abstand2 = quadrat(self.mittelpunkt.x-punkt.x) + quadrat
        (self.mittelpunkt.y - punkt.y)
        return abstand2 <= quadrat(self.radius)
```

# Beispiel: Rechtecke

- Annahme: achsenparallel

```
def Rechteck(Zwo_D_Objekt):  
    def __init__(self, basis, groesse):  
        Zwo_D_Objekt.__init__(self, "rechteck")  
        self.basis = basis  
        self.groesse = groesse  
  
    def als_text(self):  
        return "Rechteck("+self.basis.als_text()+", "+self.groesse.als_text()+")"  
  
    def enthaelt(self, punkt):  
        if punkt.x < self.basis.x: return False  
        if punkt.x > self.basis.x+self.groesse.x: return False  
        if punkt.y < self.basis.y: return False  
        if punkt.y > self.basis.y+self.groesse.y: return False  
        return True
```

# LSP: Liskov Substitution Principle

- Barbara Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices. 23(5), May 1988
- Funktionen, die Referenzen auf die Basisklasse erwarten, sollen Exemplare der Ableitung verarbeiten können, ohne es zu wissen.
- *“If for each object  $o_1$  of type  $S$  there is an object  $o_2$  of type  $T$  such that for all programs  $P$  defined in terms of  $T$ , the behaviour of  $P$  is unchanged when  $o_1$  is substituted for  $o_2$  then  $S$  is a subtype of  $T$ .”*

# “ist-ein” und “hat-ein”

- Relationen zwischen Klassen A und B:
  - A kann Basisklasse von B sein
  - A kann Element/Feld von B sein
    - genauer: Exemplare von A können Felder von Exemplaren von B sein
- “B ist ein A” (*is-a*): Exemplare von B können überall da auftreten, wo auch Exemplare von A erlaubt sind
  - Beispiel: Rechteck ist-ein Zwo\_D\_Objekt
  - Interpretation als Klassifikation: B ist Teilmenge von A
  - besser: Interpretation im Sinne von LSP
- “B hat ein A” (*has-a*): Attribute von B haben A als Typ
  - Beispiel: Kreis hat-einen Punkt (Rechteck sogar zwei)

# Klassendefinitionen in Python

`class Name:`

Inhalt

`class Name(Basisklasse):`

Inhalt

- Inhalt: Methodendefinitionen (Funktionen)
  - auch erlaubt: Variablenzuweisungen (Klassenvariablen)
  - allgemeiner: beliebiger Pythoncode
- Erzeugung von Exemplaren:
  - `o = Name(optionale Argument)`

# Methodendefinitionen

def methodenname(self, weitere parameter):

inhalt

- Erster Parameter ist immer das Objekt, an dem die Methode gerufen wird
  - Per Konvention “self” genannt
  - C++, Java, C#: Parameter wird nicht explizit in Signatur aufgeführt; heißt automatisch “this”
  - Optionale return-Anweisung zur Definition des Methodenergebnis
- Methodenaufruf:
  - o.methodenname(weitere Argumente)



# Spezialmethoden

- Werden vom Interpreter automatisch aufgerufen
  - i.d.R. Optional: Interpreter überprüft, ob Methode definiert ist, und nimmt ansonsten Standardverhalten an
- Objekt-Initialisierung (*constructor*): `__init__`
  - Für Klasse(argumente) erzeugt der Interpreter ein Exemplar o von Klasse, und ruft an diesem Exemplar dann o.`__init__(argumente)`
    - self ist das gerade entstehende Objekt
- Objekt-Finalisierung (*destructor, finalizer*): `__del__`
  - wird aufgerufen, bevor Speicher des Objekts freigegeben wird
- String-Konvertierung: `__str__`
  - wird bei `str(o)` aufgerufen, bei Ausgabe mit `print`, ...
  - anstelle von `als_text`

# Weitere Konzepte Objekt-orientierter Programmierung

- Mehrfachvererbung (Python, C++)
- Schnittstellen (*interfaces*, Java, C#)
- Properties (C#, Python)
- Integration von OOP in Ausnahmebehandlung
- Reflection/Introspection
- Parametrisierte Typen (*generic types*) (C++, C#, Java)