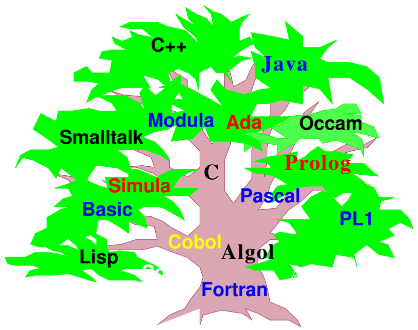
3. Grundkonzepte objektorientierter Programmierung

- Einführung
  - Historie
  - Ziele objektorientierter Software-Entwicklung
- Modularisierung, Datenkapselung, Abstrakte Datentypen
- Objekte und Klassen
- Vererbung und Klassenhierarchien
- Überladen von Funktionen (Polymorphismus)
- Generische Klassen



Objektorientierung

"Object-oriented" is the latest IN term, complementing or even replacing "structured" as the high-tech version of "good".

Bertrand Meyer, 1988

"Object-oriented" is well on its way to becoming the buzzword of the 1990s. Suddenly everybody is using it, but with such a range of radically different meanings that no one seems to know exactly what the other is saying ...

How can the same term refer to things as different as programming languages, user interfaces, schemes for representing expert knowledge, distributed operating systems, telephony switching systems, and many others.

B.J. Cox, 1991

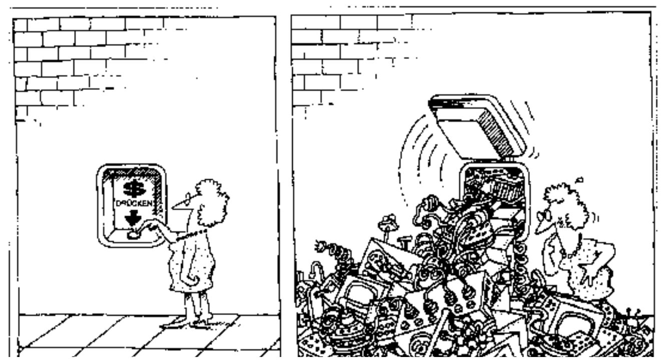
Historie

- SIMULA67
  - entwickelt von 1965-67 am Computercenter Oslo (K. Nygaard, O. Dahl)
  - Hauptneuerung: Klassenkonzept mit dynamisch erzeugbaren Instanzen
  - Unterstützung von Unterklassen
  - erste objektorientierte Programmiersprache
- Abstrakte Datentypen: CLU, Ada, Modula
- Smalltalk
  - entwickelt am XEROX Palo Alto Research Center (70er Jahre)
  - Entwicklung in Zusammenhang mit ersten grafischen Oberflächen für Workstations
  - SMALLTALK-System besteht nur noch aus Klassen mit ihren Objekten
  - vorgegebene Klassenhierarchie wird vom Anwendungsentwickler erweitert
  - Objekte kommunizieren ausschließlich über Nachrichten (Botschaften), welche zu Methodenaufrufen führen
- Hybridsprachen, vorwiegend auf Basis von Pascal oder C
  - Object Pascal, Quick Pascal, TurboPascal
  - Objective C, C++
  - Oberon (Modula-Nachfolger)
- Eiffel
- Java (1995)

Ziele objektorientierter Software-Entwicklung

- es werden Verbesserungen v.a. bezüglich des Software-Engineering angestrebt, d.h. der ingenieurmäßigen Realisierung großer, industriell einsetzbarer Programmsysteme
  - Verbesserung der Software-Qualität
  - Verbesserung der Produktivität
- Grundproblem: inhärente Komplexität großer Software-Systeme
- Ziel: bessere Handhabung der Komplexität

⇨ objektorientierte Entwurfsverfahren (Analyse und Design); objektorientierte Implementierung



Software-Qualität

- Korrektheit
- Robustheit gegenüber Ausnahmesituationen
- Erweiterbarkeit: einfache Anpassung an geänderte Anforderungen
- Wiederverwendbarkeit: einfache Nutzung bestehender Komponenten für ähnliche Anwendungssituationen
- Kompatibilität: einfache Zusammenarbeit unterschiedlicher Softwareprodukte
- Effizienz, Portabilität, Benutzerfreundlichkeit, ...

- Hauptanteil an Software-Kosten liegt in der Wartung bestehender Programme !
  - leichte Änderbarkeit kritisch
  - Voraussetzung: Software-Entwurf mit geeigneter Modularisierung

Beispiele

- Probleme bezüglich Erweiterbarkeit
  - Verwaltung von Bibliotheksobjekten durch variante Record-Struktur (Modula):

```
TYPE Ausleihobjekt = RECORD
        Invnr:      INTEGER;
        AJahr:      INTEGER;
        CASE Art: (Buch, Zeitschrift) OF
            Buch:            ISBN: ... |
            Zeitschrift:     Bandnummer: ... ;
        END  (* CASE *)
    END; (* RECORD *)
```

Zugriff auf Variable *Aobj*:

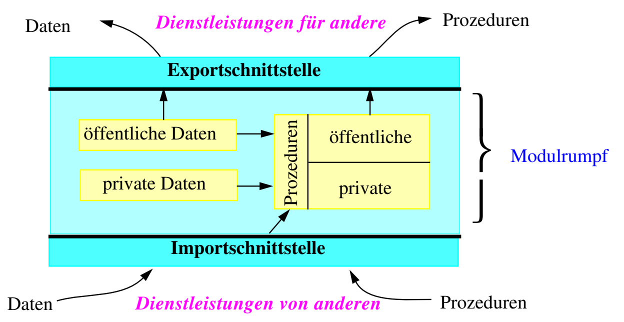
```
CASE Aobj.Art     OF
     Buch:            ... |
     Zeitschrift:     ... ;
END  (* CASE *)
```

  - zusätzliche Objektart verlangt Anpassungen in allen Programmteilen, in denen auf Bibliotheksobjekte zugegriffen wird

- Probleme bezüglich Wiederverwendbarkeit
  - gegeben seien Prozeduren zur Suche auf einem Feld von INTEGER-Werten
  - Suche soll unterstützt werden für
    - andere Elementarten,
    - variable Anzahl von Elementen,
    - zusätzliche Suchbedingungen, • • •

Modularisierung

- Zerlegung von Gesamtaufgabe bzw. Gesamtsystems in Teilaufgaben bzw. Teilsysteme (Module)
- Modul
  - Zusammenfassung von Daten (Konstanten, Datentypen, Variablen) und Prozeduren zu einer Einheit
  - Kommunikation mit anderen Moduln erfolgt nur über wohldefinierte Schnittstelle (keine Seiteneffekte)
  - Modul-Implementierung, d.h. konkrete Realisierung von Datentypen und Prozedurrümpfen, bleibt vor allen anderen Moduln verborgen



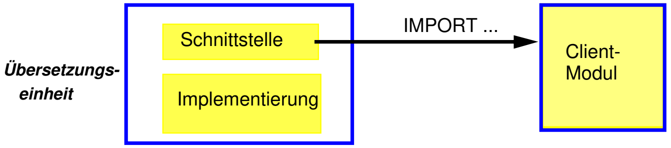
Modularisierung (2)

- Modul soll
  - klar abgetrennte Teilaufgabe des Gesamtsystems realisieren
  - unabhängig von anderen Moduln spezifizierbar, implementierbar, test- und verifizierbar sein
- Modularisierung ist Voraussetzung zur Handhabung größerer Systeme
  - unterstützt Aufteilung der Programmiertätigkeit
- Kriterien der Modularisierung nach Meyer*
  - Module müssen syntaktische Einheiten der verwendeten Sprache sein
  - jedes Modul sollte mit möglichst wenig anderen Modulen kommunizieren
  - kommunizierende Module sollten möglichst wenig Information austauschen
  - Kommunikationsbeziehungen sollten im Programmtext explizit gemacht werden
  - alle Information eines Moduls - bis auf die explizit öffentlich gemachte - sollte verborgen sein
  - Module sollten sowohl offen, als auch geschlossen sein
    - offen für Erweiterungen
    - geschlossen, d.h. zuverlässig verfügbar, für andere Module

* B. Meyer: Objektorientierte Software-Entwicklung, Hanser 1990

Module in Modula-2

- Modula-2 unterstützt separate Übersetzbarkeit von Moduln mit vollständiger Typprüfung (⇔ getrennte Übersetzbarkeit)
- Dienstmodule (Server-Module) werden zerlegt in Schnittstellenmodule (DEFINITION MODULE) und Implementierungsmodule (IMPLEMENTATION MODULE)



- Client-Module importieren benötigte Prozeduren, Typen und Variablen explizit
- Schnittstellenprüfung durch Compiler kann ohne Kenntnis der Implementierung erfolgen
- Änderungen der Implementierung erfordern keine neue Übersetzung des importierenden Moduls

Modularisierung (4)

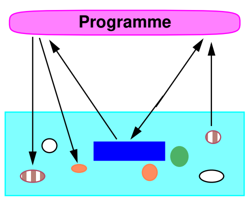
- es bestehen unterschiedliche Arten von (Dienst-) Moduln

1. Funktionsmodule
   - Sammlung von Prozeduren bzw. Funktionen
   - keine lokalen Datenobjekte (“gedächtnislos”)
2. Sammlung von Konstanten und Datentypen
3. Datenkapsel
   - Verwaltung einer oder mehrerer Datenstrukturen
   - Zugriff nur über Prozeduren (Operationen) der Modul-Schnittstelle
4. Abstrakter Datentyp (ADT)
   - benutzerdefinierter Datentyp, dessen Struktur verborgen wird
   - Operationen zum Erzeugen und Bearbeiten von Objekten dieses Typs

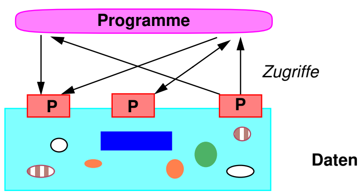
- Datenkapsel und ADT realisieren unterschiedliche Stufen von Datenabstraktion

Datenkapselung / Information Hiding

- Geheimnisprinzip: Information Hiding (D.L. Parnas, 1972)
  - Implementierung komplexer Datenstrukturen soll in einem Baustein (Modul) verborgen werden
  - Klienten erhalten nur eine abstrakte Sicht auf die Daten (abstrakte Datenstrukturen)
  - Datenzugriff nur über Prozeduren, die zum Baustein gehören
- Unkontrollierter vs. kontrollierter Datenzugriff



freier Datenzugriff



Datenzugriff über Prozeduren (P)

- Kapselung (encapsulation) beinhaltet zwei Aspekte
  - Verknüpfung von Daten mit zugehörigen Operationen
  - Geheimnisprinzip: Verbergen der Struktur von Daten sowie der Implementierung der Operationen (Information Hiding)

Datenkapselung (2)

- Gekapselte (abstrakte) Datenstrukturen besitzen einen Zustand (“Gedächtnis”), der durch Zugriffsprozeduren geändert werden kann
- Die Datenkapselung hat folgende Vorteile:
  - gekapselte Datenstrukturen sind einfacher nutzbar (reduzierte Komplexität)
  - sie können ohne Auswirkung auf andere Moduln geändert werden
  - fahrlässiger Mißbrauch ist nicht möglich, vorsätzlicher Mißbrauch ist erschwert (Zugriffsschutz)
- Es bestehen jedoch auch Nachteile
  - Zugriff etwas weniger effizient
  - eingeschränkte Flexibilität des Zugriffs, da nur über Prozeduren der Schnittstelle auf die Daten zugegriffen werden kann
- Verbergen von Daten um jeden Preis nicht sinnvoll, sondern Festlegung der Schnittstelle, so daß
  - einfache Benutzung erreicht wird und
  - Änderungen der Implementierung ohne Auswirkungen auf nutzende Module bleiben

Beispiel: Queue-Verwaltung - ungekapselt (Modula)

```
•••
TYPE QTyp = POINTER TO ElementTyp;
     ElementTyp = RECORD
        Wert: INTEGER;
        Nachfolger: QTyp;
     END (* RECORD *);
VAR Schlange: QTyp;
     i: INTEGER;

PROCEDURE DeQueue(VAR x: INTEGER);
VAR P, VorP: QTyp;
BEGIN (* letztes Element entfernen *)
 IF Schlange <> NIL THEN
     P := Schlange; VorP:= NIL;
     WHILE P^.Nachfolger <> NIL DO
            VorP := P; P := P^.Nachfolger;
     END (* WHILE *);
     x := P^.Wert;
     DISPOSE (P);
     IF VorP = NIL THEN
        Schlange:= NIL
     ELSE
        VorP^.Nachfolger := NIL;
     END (* IF *);
 END (* IF *);
END DeQueue;
```

```
PROCEDURE EnQueue(x: INTEGER);
     VAR Neu: QTyp;
BEGIN NEW (Neu);
     Neu^.Wert := x;
     Neu^.Nachfolger := Schlange;
     Schlange := Neu;
END EnQueue;


BEGIN

Schlange := NIL;
FOR i := 1 TO 10 DO
     EnQueue (2*i);
END;
WHILE Schlange <> NIL DO
     DeQueue (i);
     WriteInt (i, 5);
END;
•••
```


Beispiel: Queue-Verwaltung - gekapselt (Modula)

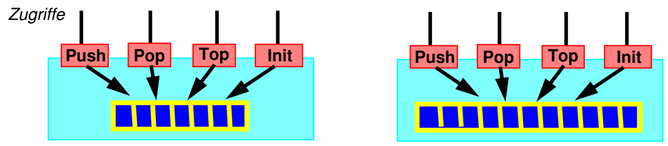
```
DEFINITION MODULE Queue
(*  FIFO-Speicher für INTEGER-Zahlen *)
PROCEDURE EnQueue (x: INTEGER);
PROCEDURE DeQueue (VAR x: INTEGER);
PROCEDURE isEmpty (): BOOLEAN;
PROCEDURE isFull (): BOOLEAN;
END Queue.
```

```
MODULE QueueClient;
FROM InOut IMPORT (* PROC *) WriteInt;
FROM Queue IMPORT (* PROC *) EnQueue,
              DeQueue, isEmpty, isFull;
VAR i: INTEGER;

BEGIN
   •••
   FOR i := 1 TO 10 DO
      EnQueue (2*i);
   END;
   WHILE NOT isEmpty () DO
      DeQueue (i);
      WriteInt (i, 5);
   END;
   •••
END QueueClient.
```

```
IMPLEMENTAION MODULE Queue;
TYPE QTyp = POINTER TO ElementTyp;
     ElementTyp = RECORD
        Wert: INTEGER;
        Nachfolger : QTyp;
     END (* RECORD *);
VAR Schlange: QTyp;

PROCEDURE EnQueue(x: INTEGER);
     • • •
END EnQueue;

PROCEDURE DeQueue(VAR x: INTEGER);
     • • •
END DeQueue;

PROCEDURE isEmpty (): BOOLEAN;
BEGIN RETURN (Schlange = NIL)
END isEmpty;

PROCEDURE isFull (): BOOLEAN;
BEGIN RETURN FALSE
END isFull;

BEGIN (* Queue-Initialisierung *)
   Schlange:= NIL;
END Queue.
```


Abstrakte Datentypen

- ADT
  - **Datentyp**, dessen innere Struktur nicht sichtbar ist
  - Zugriff auf Objekte dieses Typs nur mittels **zulässiger Operationen**
- Beispiel: vordefinierter ADT **INTEGER**
  - Repräsentation ist verborgen
  - Zugriff nur über vordefinierte Operationen: +, -, *, DIV, MOD, =, <>, >, ...
- jetzt: Erweiterung auf benutzerdefinierte ADTs
- Bsp.: ADT **STACK**, Operationen: Push, Pop, Top, Init, ...



Stack S1 Stack S2

Abstrakte Datentypen (2)

- Hauptunterschied zur Datenkapselung:
  ADT erlaubt beliebig viele (gekapselte) Instanzen bzw. Variablen dieses Typs
- Konsequenzen
  - Initialisierung muß nun für jedes Objekt einzeln erfolgen: aufrufbare Konstruktor-Operation
  - ggf. Operation zur Beseitigung von Objekten (Speicherfreigabe): Destruktor
- benutzerdefinierte ADTs können wie vordefinierte Datentypen verwendet werden
  - ⇨ Erweiterung der Sprache um neuen Datentyp
  - ⇨ bessere Eignung zur Lösung eines bestimmten Problems
- Vorteile des ADT-Konzepts
  - wie bei Datenkapselung: reduzierte Komplexität, einfache Änderbarkeit, Zugriffsschutz
  - beliebige Anzahl von ADT-Instanzen erhöht Einsetzbarkeit
  - bessere Wiederverwendbarkeit für unterschiedliche Anwendungen

Abstrakte Datentypen in Modula-2

- Realisierung durch Modulkonzept (Trennung zwischen Schnittstelle und Implementierung)

Schnittstelle:

TYPE **pT**; (* privater Typ *)
Prozedurdeklarationen
(auch Initialisierungsprozedur)

Implementierung:

TYPE pT = ...;
Prozedurdefinitionen
interne Daten und Prozeduren
(* keine Initialisierung *) END

→ IMPORT: privater Typ, Prozeduren

Kundenmodul:

VAR ...: **pT**
Prozeduraufrufe an ADT
(insbesondere von Konstruktor und Destruktor)

- Verwendung privater (opaker) Typen, deren Aufbau unsichtbar ist
- Kundenmoduln importieren den opaken Typ samt den Prozeduren und legen Variablen an
- Alle Operationen (inkl. Initialisierung, Vergleiche und Wertzuweisungen) erfolgen durch den Modul, der den ADT exportiert. Die betreffenden Variablen werden als Parameter übergeben
- explizite Initialisierung der Variablen durch eigene Prozedur (Konstruktor)

ADT Warteschlange (Queue) in Modula

```
DEFINITION MODULE QueueADT;
TYPE QTyp;     (* privater/opaker Typ   *)
PROCEDURE EnQueue (VAR q: QTyp; x: INTEGER);
PROCEDURE DeQueue (VAR q: QTyp;VAR x: INTEGER);
PROCEDURE isEmpty (q: QTyp): BOOLEAN;
PROCEDURE isFull (q: QTyp): BOOLEAN;
PROCEDURE QInit (VAR q: QTyp);(* Konstruktor *)
PROCEDURE QDelete(VAR q: QTyp);(* Destruktor *
END QueueADT.
```

```
MODULE QueueClient;
FROM InOut IMPORT WriteInt, WriteLn;
FROM QueueADT IMPORT (* PROC *) EnQueue,
      DeQueue, isEmpty, isFull, QInit,
      QDelete,(* TYPE *) QTyp;
VAR Q1, Q2: QTyp; •••
BEGIN
  QInit (Q1); QInit (Q2);
  FOR i := 1 TO 10 DO
       EnQueue (Q1, i);EnQueue (Q2, 2*i);
  END;
  WHILE NOT isEmpty (Q1) OR isEmpty (Q2) DO
       DeQueue (Q1, i); DeQueue (Q2, j);
       WriteInt (i, 5); WriteInt (j, 5);
       WriteLn;
  END;
  QDelete (Q1); QDelete (Q2); •••
END QueueClient.
```

```
IMPLEMENTAION MODULE QueueADT;
TYPE  QTyp = POINTER TO ElementTyp;
      ElementTyp = • • •
  PROCEDURE EnQueue(VAR q: QTyp;
                        x: INTEGER);
                  • • •
     • • •

  PROCEDURE QInit (VAR q: QTyp);
  BEGIN q := NIL
  END QInit;

     (* leerer Rumpf *)
END QueueADT.
```


Schwächen von ADTs in Modula

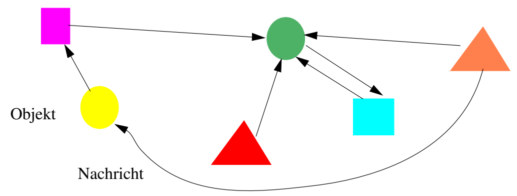
- Module sind keine Typen
  - Modul kann beliebig viele Definitionen von Typen, Variablen und Prozeduren enthalten
  - Daten und Operationen bilden innerhalb eines Moduls keine syntaktische Einheit
  - neben Typen müssen zugehörige Operationen explizit importiert werden
  - opake (private) Typen stellen Behelfsansatz zum Verbergen der Typstruktur dar
- nur Pointer-Typen können als ADTs verwendet werden
- ADT-Instanzen können durch allgemeine Pointer-Operationen (Zuweisung, Vergleich) manipuliert werden

```
QInit (Q1); QInit (Q2);
EnQueue (Q1, 2); EnQueue (Q2, 3); ...
Q1 := Q2; (* ursprüngliche Queue Q1 stellt Garbage dar *)
DeQueue (Q1, c);
```

- korrekte Initialisierung kann nicht gewährleistet werden
- keine parametrisierbaren (generische) ADTs
- Semantik der Operationen nicht festgelegt (Ansätze dazu finden sich in Eiffel)
- unzureichende Unterstützung der Wiederverwendbarkeit

Objekte und Klassen

- Modularisierung der Software in Menge von Objekten



- Objekte umfassen Daten und Operationen (Prozeduren, Methoden)
- gleichartige Objekte bilden eine (Objekt-) Klasse
- grobe Begriffsentsprechungen:

  Klasse - **Abstrakter Datentyp**

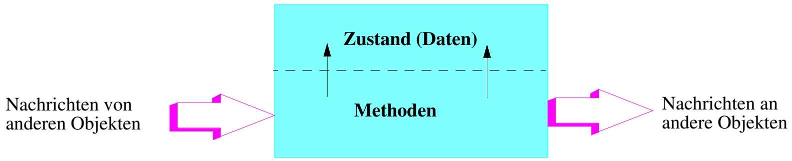
  Objekt - **ADT-Instanz** (Klasseninstanz)

- "rein" objektorientierte Sprachen erzwingen Verwendung von Objekten sowie vollständige Kapselung

Objekte

- Objekte besitzen
  - *Zustand*: Wert von Zustandsvariablen (Attributen, Instanzvariablen, Datenelementen)
  - *Verhalten*: Methoden (Operationen, Instanzmethoden, Elementfunktionen), bestehend aus Schnittstelle (Signatur) und Implementierung (Rumpf)

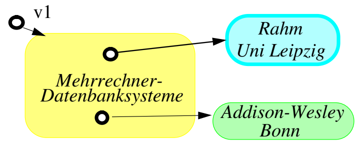
  und sind einer Klasse zugeordnet



- Kommunikation zwischen Objekten über Nachrichten (Methodenaufrufe)
- Kapselung
  - Objektzustand ist nur für Methoden des Objektes sichtbar und änderbar
  - Methodenimplementierung ist für andere Objekte unsichtbar

Objekt-Referenzen

- Zustandsvariablen von Objekten können enthalten
  - einfache Werte
  - strukturierte Werte wie Tupel-Werte (Record-Strukturen) oder Mengen-Werte
  - Referenzen (Verweise) auf andere Objekte
- durch Verwendung von Objektreferenzen als Objektwerte können komplexe Objekte realisiert werden, welche Teilobjekte als Komponenten besitzen



- Verwendung von Objekt-Referenzen erlaubt **Objekt-Sharing ohne Redundanz !**
- Referenzen (Verweise) vs. Zeiger (Pointer)
  - Referenzen dienen zur Bezeichnung von Objekten unabhängig von der konkreten Objektspeicherung
  - Zeiger sind (logische) Speicheradressen und repräsentieren dynamisch allokierte Speicherplätze auf der Halde
  - Zeiger stellen eine Möglichkeit dar, Referenzen zu implementieren

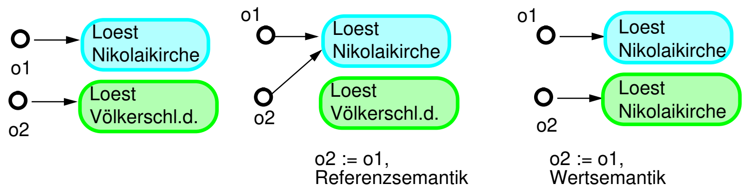
Identität vs. Gleichheit von Objekten

- Objektidentität:
  zwei Objekte o und o' sind identisch, o == o', wenn sie dieselbe Referenz aufweisen und somit denselben Speicherplatz belegen
- Objektgleichheit:
  zwei Objekte o und o' sind gleich, d.h. o = o', wenn sie denselben Zustand aufweisen

- unterschiedliche Sprachunterstützung
  - Simula, Smalltalk: Unterscheidung zwischen Identitätstest (==) und Gleichheitstest (=)
  - Eiffel: Identitätstest (=)
  - C++: Gleichheitstest == auf Objekten nicht definiert (kann jedoch überladen werden)

Objektzuweisung

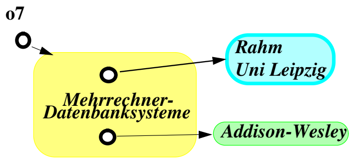
- Zuweisungen von Objekten *o1 := o2* können auf unterschiedliche Weise realisiert werden
- Referenzsemantik: o1 und o2 sind nach der Zuweisung identisch
- Wertsemantik: der Zustand von o2 wird kopiert, so daß o1 und o2 gleich sind



- unterschiedliche Sprachunterstützung
  - Eiffel unterstützt beide Zuweisungssemantiken (Wertsemantik für Standard-Datentypen, sonst Referenzsemantik)
  - Smalltalk: nur Referenzsemantik
  - C++ defaultmäßig Wertsemantik

Kopieren von Objekten

- bei Wertsemantik ist für komplexe Objekte flaches oder tiefes Kopieren möglich
- flaches Kopieren (*shallow copy*): es wird nur das betroffene Objekt mit seinen Referenzen kopiert
- tiefes Kopieren (*deep copy*): rekursives Kopieren aller Komponentenobjekte



- zwei Arten von Gleichheit: flache Gleichheit (*shallow equality*) und tiefe Gleichheit (*deep equality*)
- Smalltalk bietet für alle Klassen Methoden `deepCopy` und `shallowCopy`;
  C++ bietet defaultmäßig flaches Kopieren

Klasse

- Objekte mit gleicher Struktur und gleichem Verhalten werden zu Klassen zusammengefaßt
- Klasse legt Typ der Objekte fest
- Klassendeklaration beschreibt
  - Zustandsvariablen (Name, Typ, Sichtbarkeit)
  - Methodenköpfe (Name, Parameter, Sichtbarkeit)
  - Methodenrümpfe (verborgen)
  - ggf. Klassenvariablen
  - ggf. Klassenmethoden
- Beispiel:

```
Klasse STUDENT
  public: Immatrikulation (Name, Datum, ...);
          Exmatrikulation ();
          Gib_Matnr (): integer;
          Gib_AnzahlStudenten (): integer;
  Instanzvariablen
          SName ...
          ImmDatum ...
          MatNr ...
  Instanzmethoden
          Immatrikulation ...
  Klassenvariablen
          SAnzahl: integer;
  Klassenmethoden
          Gib_AnzahlStudenten ... return SAnzahl ...
```


Beispiel: Queue-Verwaltung (C++)

Schnittstelle (Header-Datei): *queue.h*

```
struct QElement {     int Wert;
                      QElement* next; };
class Queue
{ public:  Queue ();      // Konstruktor
       void EnQueue (int);
       int DeQueue ();
       int IsEmpty ();
private: QElement *first;};
```

Hauptprogramm:

```
#include "queue.h"
#include <stdio.h>
main () {
Queue Q1, Q2;
int i;
   for (i=1; i < 10; ++i)
     { Q1.EnQueue (i); Q2.EnQueue (i*2); }
   while (! (Q1.IsEmpty () | Q2.IsEmpty () ))
      printf ("Q1: %2i   Q2: %2i\n",
              Q1.DeQueue (), Q2.DeQueue ());
}
```

Implementierung:

```
#include "queue.h"

Queue::Queue ()     // Konstruktor
   { first = 0; /* NULL */ }

void Queue::EnQueue (int i)
{  QElement *help;
   help  = new QElement;
   help -> Wert = i;
   help -> next = first;
   first = help; }

int Queue::DeQueue ()
{ QElement  *p, *p1;
   int x;
if (first != 0)
{  p = first; p1=0;
   while (p -> next != 0)
       {p1 = p; p = p -> next; }
   x = p -> Wert;
   delete p;
   if (p1 == 0) first = 0;
       else p1 -> next=0;
   return x; }
}

int Queue::IsEmpty ()
   // liefert TRUE (1) falls first=0
{ return ! first ; }
```


Methoden

- im Gegensatz zu allgemeinen Prozeduren sind Methoden an eine Klasse gebunden
- jeder Methodenaufruf (jede Nachricht) hat ein Objekt als Empfänger

  Bsp.: Q2.DeQueue ()

- allgemeines Nachrichtenformat:

Objekt	Selektor	Argumente
Empfänger	bestimmt auszuführende Methode	

- Methodenkopf legt die Menge der Nachrichten fest, die von Methode verarbeitet werden können
- Methodenrumpf legt Verhalten des Objektes beim Empfang einer zulässigen Nachricht fest
- Nachricht an Empfängerobjekt innerhalb einer Methode erfolgt mit spezieller Bezeichnung
  - C++, Java: `this`
  - Eiffel: `CURRENT`
  - Smalltalk: `self`

Methoden (2)

- Methoden können folgendermaßen klassifiziert werden
  - Änderungsfunktionen (Mutatoren): bewirken Änderung des Objektzustands
  - Beobachter: lesender Zugriff auf Objektzustand
  - Konstruktoren: Generierung und ggf. Initialisierung neuer Objekte
  - Destruktoren: Löschen von Objekten

  - Instanz- vs. Klassenmethoden
- Konstruktoren:
  - erzeugen neues Exemplar der Klasse mit zunächst undefiniertem Zustand
  - gleichzeitige (parametrisierte) Zustands-Initialisierung oder separat
  - statisch oder dynamisch erzeugte Objekte

  - Smalltalk: Standard-Methode `new`
  - Eiffel: Konstruktor CREATE dient auch zur Initialisierung
  - C++: für jede Klasse *K* existiert Default-Konstruktor (ohne Parameter) namens *K*, der jedoch redefiniert werden kann (z.B. zur Initialisierung)
  - C++, Oberon: statisch und dynamisch erzeugte Objekte; Eiffel, Smalltalk: nur dynamisch
- Freigabe von Objekten: explizit durch Destruktoraufruf (C++, Name ~*K* für Klasse *K*) oder automatisch durch “garbage collector” (Smalltalk, Eiffel)

Methoden (3)

- vollständige Kapselung erzwingt Methodenaufrufe für sämtliche Objektzugriffe
  - Bereitstellung von Lese- und Änderungsmethoden für Zustandsvariablen (Attribute)
  - Änderung der Attributrepräsentation bleibt ohne Auswirkungen auf nutzende Programme
- Objektkapselung variiert jedoch für verschiedene Sprachen:
  - Smalltalk: alle Zustandsvariablen sind verborgen, die Schittstellen aller Methoden sind sichtbar
  - C++: Zustandsvariable und Methoden können selektiv als öffentlich (public), privat (private) oder geschützt (protected) deklariert werden

  Tradeoff: Unabhängigkeit vs. Flexibilität
- Smalltalk: Zustandsvariable bzw. Werte werden auch als Objekte und Zugriffe auf sie als Methodenausführungen aufgefaßt

  Bsp.: mwert <- mwert + 1

Objekterzeugung und -freigabe in C++

- in C++ werden Objekte entweder statisch deklariert oder dynamisch erzeugt

	statisch	dynamisch
Objektdeklaration	`Queue q;`	`Queue* qp;`
Objekterzeugung	*implizit*	`qp = new Queue ();`
Methodenaufruf	`q.EnQueue (i);`	`qp -> EnQueue (i);`
Attributzugriff	`i = q.AnzahlElemente;`	`i = qp -> AnzahlElemente;`
Objektfreigabe	*implizit am Prozedurende*	`delete qp;`

- Konstruktoren werden automatisch aufgerufen, wenn ein Objekt erzeugt wird
  - für dynamisch erzeugte Objekte im Rahmen der `new`-Anweisung
  - für statische Objekte zum Aufrufzeitpunkt der Prozedur, welche die Deklaration enthält
  - für eine Klasse kann es mehrere Konstruktoren mit unterschiedlichen Parameterlisten geben
- Aufruf des Destruktors (falls vorhanden): bei `delete`-Operation (unmittelbar vor Speicherfreigabe)
- Konstruktoren und Destruktoren dürfen keinen Rückgabetyp besitzen (auch nicht `void`)

Beispiel: Objekterzeugung und -freigabe (C++)

Schnittstelle (Header-Datei): *polygon.h*

```
class POLYGON
 { public:
     POLYGON ();  // Default-Konstruktor
     POLYGON (int);    // Konstruktor
     ~POLYGON ();  // Destruktor
     double Flaeche ();
   private: int AnzahlEcken;
            static int AnzPolygone;
                  // Klassenvariable
 };
```

Hauptprogramm:

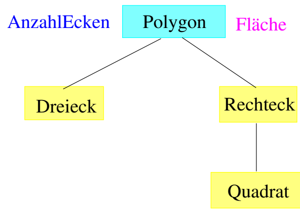
```
#include "polygon.h"
main () {
   POLYGON p1, p2 (3), *p3, *p[8];
   int i;
   p3 = new POLYGON (4);
   for (i=0; i < 8; ++i)
        p[i] = new POLYGON (i+3);
  for (i=7; i>=0; --i) delete p[i];
}
```

Implementierung:

```
#include "polygon.h"
#include <iostream.h>

POLYGON::POLYGON () { ++AnzPolygone; }
             // Default-Konstruktor
POLYGON::POLYGON (int n)
   { AnzahlEcken = n;
    cout    << " neues Polygon mit "
            << AnzahlEcken
            << " Ecken " << endl;
      ++AnzPolygone;
   }

POLYGON::~POLYGON () // Destruktor
{ --AnzPolygone;
cout << "Loeschen eines Polygons mit "
        << AnzahlEcken << " Ecken; "
   << AnzPolygone
   << " Polygone verbleiben noch"
   << endl;
   }

double POLYGON::Flaeche ()   {••• }

int POLYGON::AnzPolygone = 0;
      // Statische Klassenvariable
```

Vererbung

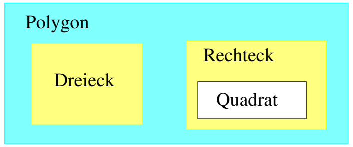
- grundlegendes Konzept objektorientierter Systeme
- Erweiterung bestehender Objektklassen durch Bildung von Subklassen (Unterklassen)
  - alle Eigenschaften (Attribute, Methoden-Schnittstelle und -Implementierung) können an Unterklasse vererbt werden
  - Unterklasse kann zusätzliche Eigenschaften aufweisen
  - Methoden können in Unterklassen überschrieben werden (overriding)
  - Subklassenbildung entspricht einer *Spezialisierung* der übergeordneten Klasse



- umgekehrte Sichtweise: gemeinsame Eigenschaften verschiedener Klassen können innerhalb einer Oberklasse (Superklasse) zusammengefaßt werden (*Generalisierung*)
- unterstützt Wiederverwendbarkeit

Vererbung (2)

- einfache Vererbung (single inheritance): jede Klasse hat höchstens eine Oberklasse
  => es liegt eine Typhierarchie vor
- Is-A-Semantik: Objekte einer Subklasse sind auch Objekte der übergeordneten Superklassen
  - Teilmengenbeziehung zwischen Unter- und Oberklasse
  - ein Objekt kann mehrere "Rollen" annehmen



- Unterklassen können jedoch ggf. eingeschränkt werden (z.B. in C++), so daß nicht alle Methoden und Attribute der Oberklasse verfügbar sind

Beispiel: Vererbungshierarchie in C++

Schnittstelle (Header-Datei): *polygon.h*

```
class POLYGON
 { public:
    POLYGON ();  // Default-Konstruktor
    POLYGON (int);    // Konstruktor
    ~POLYGON ();  // Destruktor
  private: int AnzahlEcken;
           static int AnzPolygone;
                 // Klassenvariable
 }
class DREIECK : public POLYGON
 { public:
  DREIECK (); // Default-Konstruktor
 };
class RECHTECK : public POLYGON
 { public: RECHTECK (double, double);
                 // Konstruktor
           double Flaeche ();
   private: double Breite, Hoehe;
 };

class QUADRAT : public RECHTECK
 { public: QUADRAT  (double);
                 // Konstruktor
 };
```

Implementierung:

```
#include "polygon.h"
#include <iostream.h>

// POLYGON-Implementierung wie zuvor
•••
RECHTECK::RECHTECK (double h, double b):
        POLYGON (4)    // Konstruktor
    {  Hoehe = h; Breite = b; }

double RECHTECK::Flaeche ()
    { return Hoehe * Breite;  }

QUADRAT::QUADRAT (double h):
        RECHTECK (h,h) {}
  // erzwingt Quadrateigenschaft
```

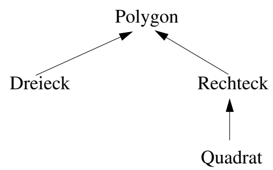
Hauptprogramm:

```
#include "polygon.h"
#include <iostream.h>
main () {
  DREIECK d;
  RECHTECK r (5.5, 4.3);
  QUADRAT q (2.5);
  cout  << "Flaeche r: " << r.Flaeche()
        << endl;
  cout  << "Flaeche  q: " << q.Flaeche()
        << endl;
}
```


Substituierbarkeitsprinzip

- Objekte einer Unterklasse können generell für Objekte der Oberklasse substituiert werden (jedoch nicht umgekehrt)
  - Methoden der Oberklasse sind auch für Objekte einer Unterklasse ausführbar
  - Zuweisungsregel: Objekten (Variablen) einer Oberklasse können Objekte von Unterklassen zugewiesen werden
- Beispiele für Zuweisungskompatibilität

```
POLYGON p; RECHTECK r; DREIECK d; QUADRAT q;

Welche Zuweisungen sind erlaubt ?

  a) p = r;     b) r = p;
  c) p = d;     d) p = q;
  e) r = d;     f) d = r;
  g) r = q;     h) d = q;
```



- Substituierbarkeitsprinzip impliziert, daß Variable Objekte unterschiedlicher Klassen (Typen) aufnehmen kann (-> Polymorphismus)
  - *statischer Typ:* die bei Deklaration einer Variablen angegebene Klasse
  - *dynamischer Typ*: der sich zur Laufzeit aufgrund von Zuweisungen ergebende Typ (kann Unterklasse des statischen Typs sein)

Polymorphismus

- griechisch für "viele Formen"
- mehrere Erscheinungsformen
  - Variablen: Möglichkeit, unterschiedliche Typen anzunehmen (-> dynamischer Typ)
  - Methoden: unterschiedliche Implementierungen mit gleichem Namen (-> Überladen)
  - Operatoren: Anwendbarkeit auf unterschiedliche Operandentypen
  - generischer Polymorphismus: parametrisierte Klassen (-> Generizität)
- Überladen (Overloading): derselbe Methodenname wird für mehrere Prozeduren verwendet
- Overriding: spezielle Form des Overloading innerhalb von Klassenhierarchien
  - Redefinition von Methoden für Unterklassen (Überschreiben, Overriding)
  - spezialisierte Methode mit gleichem Namen und i.a. gleicher Schnittstelle
- Verwendung eines einzigen Methodennamens erleichtert Realisierung nutzender Programme und verbessert Software-Wiederverwendbarkeit
- Overriding impliziert *dynamisches Binden* zur Laufzeit (late binding)
  - die Nachricht *o.M* bewirkt den Aufruf der Methode *M*, die zum dynamischen Typ von *o* gehört

Overriding

- zwei Varianten: Ersetzung oder Verfeinerung von Methoden
- Ersetzung
  - Implementierung einer Methode wird durch eine neue völlig ersetzt
  - in allen objekt-orientierten Programmiersprachen möglich
- C++ verlangt, daß Methoden als "virtuell" gekennzeichnet werden, falls sie in einer Unterklasse ersetzt werden
  - dynamisches Binden erfolgt nur für virtuelle Methoden !
  - erfordert i.a. Verwendung dynamischer Objekte

```
class POLYGON
 { public:
    virtual double Flaeche ();
    ••• }

class RECHTECK : public POLYGON
 { public: double Flaeche (); •••
 };
class QUADRAT : public RECHTECK
 { ••• };
```

```
double POLYGON::Flaeche ()
   { return 0.0;  }

double RECHTECK::Flaeche ()
   { return Hoehe * Breite;  }
```

```
POLYGON* p1 = new RECHTECK r (5.5, 4.3);
POLYGON* p2 = new QUADRAT q (2.5);
double d;
d = p1->Flaeche()+ p2->Flaeche();
```


Overriding (2)

- Verfeinerung einer Methode
  - neue Methode weist "ähnliche" Funktion auf wie für Oberklasse
  - Duplizierung der Implementierung soll verhindert werden
- Ansatz: in der neuen Implementierung wird Methode der Oberklasse aufgerufen
  - Änderung der genutzten Methode wird umgangen
  - Implementierung der genutzten Methode muß nicht bekannt sein
- C++: Zugriff auf Methode der Oberklasse O erfolgt über Qualifikator *O::*

```
class POLYGON
 { public:  void virtual ShowInfo ();
    •••}

class RECHTECK : public POLYGON
 { public:  void ShowInfo ()•••
 };
```

```
void virtual POLYGON::ShowInfo() {
  cout << " Polygon mit " << AnzahlEcken
       << " Ecken " << endl; }

void RECHTECK::ShowInfo() {
       POLYGON::ShowInfo();
       cout << " Rechteck-Fläche: " <<
              Flaeche() << endl;}
```

```
POLYGON* p = new RECHTECK(5.5,4.3);
 p -> ShowInfo();
```


Overriding (3)

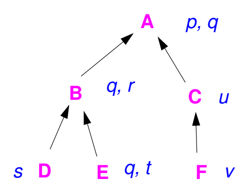
- manche objekt-orientierte Programmiersprachen erlauben nicht nur Überschreiben der Implementierung, sondern auch Überschreiben (Verfeinerung) der Methoden-Schnittstelle durch
  - Neudefinition der Datentypen von Argumenten (Parametern)
  - Neudefinition von Ergebnistypen
- Zur Wahrung der Typsicherheit sind bei Ändern der Schnittstelle folgende Regeln wichtig
  - Argumenttypen der verfeinerten Operationen müssen stets Oberklassen (oder gleich) der ursprünglichen Klasse sein (Kontravarianz)
  - der neue Ergebnistyp muß stets einer Unterklasse (oder der gleichen Klasse) angehören (Kovarianz)
- Beispiel (Kontravarianz)

für RECHTECK sei Funktion `Groesser (RECHTECK)` definiert;
für Subklasse QUADRAT ist Redefinition `Groesser (QUADRAT)` unzulässig

```
RECHTECK r1, r2;         QUADRAT q;
r1 = q;
if r1.Groesser (r2) ... ;
```

Bestimmung der Methodenimplementierung

- Problem: zu Nachricht `o.M` muß richtige Implementierung für *M* gefunden werden
- Ausgangspunkt: dynamischer Typ `D` zu o
- Smalltalk:
  - zu jeder Klasse wird in einer Methodentabelle vermerkt, für welche Methoden eine eigene Implementierung vorliegt
  - falls *M* in der Methodentabelle von D vorliegt, wird die zugehörige Implementierung ausgeführt
  - anderenfalls wird in der Oberklasse von D die Suche rekursiv fortgesetzt, bis eine Implementierung für *M* gefunden ist
  - wird keine Implementierung gefunden, erfolgt eine Fehlermeldung ("message not understood")
    -> bei statischer Typprüfung nicht möglich



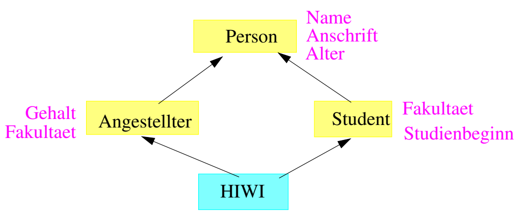
- C++ verwendet andere Implementierung:
  - zu jeder Klasse wird vollständige Methodentabelle geführt, in der auch für geerbte Methoden auf die zugehörige Implementierung verwiesen wird
  - Implementierung kann direkt mit der Methodentabelle von D bestimmt werden

Abstrakte Klassen

- Abstrakte Klassen dienen nur zur Strukturierung der Vererbungshierarchie
  - keine direkte Instantiierung von Objekten
  - Festlegung von Methoden-Schnittstellen ohne Implementierung (*abstrakte Methoden*)
  - Implementierung für abstrakte Methoden erfolgt in den Unterklassen, d.h., sie müssen überschrieben werden
- abstrakte Klassen haben wenigstens eine abstrakte Methode
- abstrakte Methoden
  - unterstützen Schnittstellen-Vererbung, keine Vererbung der Implementierung
  - ausreichend für flexible Erweiterbarkeit der Klassenhierarchie
    (Code-Vererbung nicht nötig für Erweiterbarkeit, sondern nur für Wiederverwendbarkeit)
- Definition von Attributen in abstrakten Klassen meist nicht sinnvoll
- Beispiel: abstrakte Klasse `GEOOBJEKT` mit abstrakten Methoden `Display`, `Move`, `Rotate` ...
- C++: abstrakte Methoden werden als *rein virtuelle Funktionen* deklariert
  - Beispiele: `virtual void Display () const = 0;`
  - Klassen mit wenigstens einer solchen Funktion sind abstrakte Klassen
  - es dürfen keine Objekte von ihnen angelegt werden

Mehrfachvererbung (multiple inheritance)

- Mehrfachvererbung: jede Klasse darf mehrere direkte Oberklassen besitzen
  => Klassenverband (statt Hierarchie)
  - Zyklen werden nicht zugelassen
  - Vererbung erfolgt von allen Oberklassen
- erlaubt häufig bessere Modellierung der Anwendungsgegebenheiten



- einige objekt-orientierte Programmiersprachen erlauben keine Mehrfachvererbung (Smalltalk, Oberon)

Mehrfachvererbung (2)

- bei der Vererbung kann es zu Namenskonflikten kommen
  - mehrere Oberklassen weisen Attribute oder Methoden gleichen Namens auf
  - Beispiel: `Fakultaet` in `Angestellter` und `Student`
- mögliche Auflösungsstrategien
  - *Ausschlußstrategie*: Konflikte werden per Definition ausgeschloßen (Zurückweisung von Klassendefinitionen, die zu Namenskonflikten führen) -> Umbenennung
  - *Default-Strategie*: Auswahl aus den verfügbaren Alternativen gemäß vordefinierter Regeln
  - *Qualifikationsstrategie*: explizite Angabe der Oberklasse, von welcher geerbt werden soll

  bei Methoden sind noch folgende Lösungen möglich:
  - explizite Angabe der Oberklasse bei der Benutzung für Methodenauswahl (C++)
  - Methode wird in der Subklasse überschrieben
- weitere Probleme
  - häufig komplexe und unübersichtliche Beziehungsstrukturen
  - Gefahr, daß viele Methoden und Attribute unnötigerweise geerbt werden
  - Attribute einer Klasse, von der auf mehreren Wegen geerbt wird, werden ggf. mehrfach angelegt (C++)
  - aufwendigere Methodenbestimmung

Mehrfachvererbung (3)

- Umbenennung in Eiffel

```
CLASS HiWi
    INHERIT Student RENAME Fakultaet AS SFakultaet
    INHERIT Angestellte RENAME Fakultaet AS AFakultaet
    •••
```

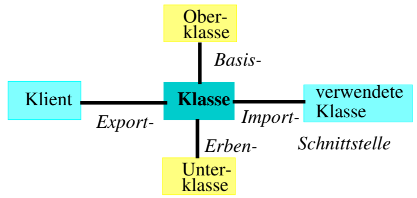
- Qualifikationsstrategie in C++

```
class HiWi: public Angestellte, public Student
    {
    ••• }
```

```
HiWi h;
•••
cout << h.Student::Fakultaet();
cout << h.Angestellte::Fakultaet();
```


Zugriffsrechte

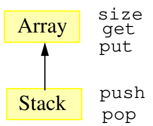
- Schnittstellen einer Klasse:



- Vererbung beeinträchtigt Kapselung -> Abhängigkeiten gegenüber Änderungen in der Klasse
- C++ bietet abgestufte Kontrolle der Sichtbarkeit von Methoden und Attributen
  - öffentlich (für Klienten zugänglich) -> PUBLIC
  - für Erben (Subklassen) zugänglich -> PROTECTED
  - geheim (nur klassenintern verwendbar) -> PRIVATE
  - einzelne Komponenten können daneben bestimmten Klienten (friends) zugänglich gemacht werden
- C++ unterstützt daneben verschiedene Vererbungsarten auf Klassenebene
  - PUBLIC: Zugriffsrechte geerbter Komponenten bleiben unverändert
  - PROTECTED bzw. PRIVATE: Zugriffsrechte werden auf PROTECTED (PRIVATE) beschränkt

Zugriffsrechte (2)

- Beschränkungen können aufgrund dynamischen Bindens jedoch teilweise umgangen werden
- Beispiel: Stack als Subklasse von Array (Stack soll Operationen `get` und `put` nicht nutzen dürfen)
  - eine solche Einschränkung der Schnittstelle wird z.B. in Eiffel unterstützt
  - in C++ auch erreichbar über geschützte Vererbung




```
array a;
stack s;

s.push (5);
s.get (1); // unzulässig
a = s;
a.get (1) // zulässig !
```

- Alternativen:
  - überschreiben der Methoden und explizite Invalidierung, z.B. durch Ausgabe einer Fehlermeldung
  - Verbessern der Vererbungshierarchie, so daß keine unerwünschten Methoden geerbt werden (Beispiel: Stack ist keine Subklasse von Array, sondern nutzt Array als Komponententyp)

Generische Klassen

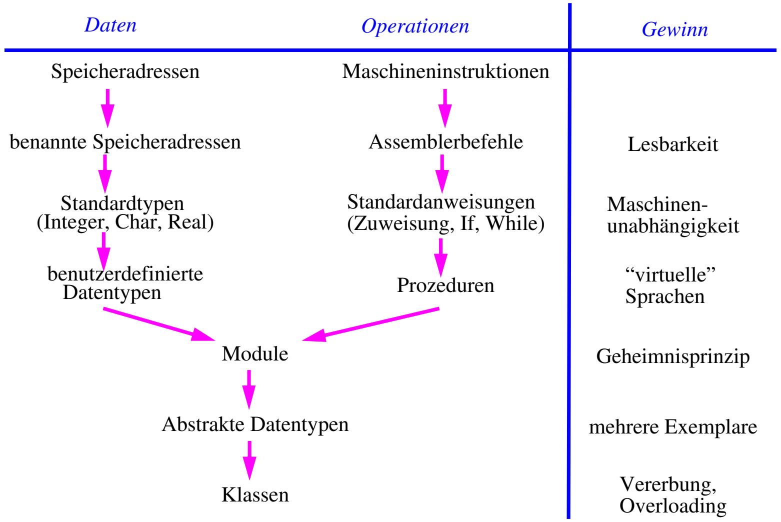
- *Generizität:* Fähigkeit, Klassen durch einen Typ zu parametrisieren
  => generische Klassen, parametrisierte Klassen, Klassen-Muster (Templates)
- Beispiel: Kollektionsklassen wie Listen, Arrays, Mengen, FIFO-Queues, Stacks etc. mit beliebigem Komponententyp
- Unterstützung in Ada, C++, Eiffel
- Beispiel: Queue-Template in C++

```
template <class T> class QUEUE
 { public:
    void EnQueue (T);
    T DeQueue ();
    QUEUE(); // Konstruktor
    ~QUEUE (); // Destruktor
private     T* Elemente
            int Anzahl}
```

```
QUEUE<int> iq;        // Integer-Queue
QUEUE<double> rq;     // Real-Queue
QUEUE<POLYGON> pq;    // Polygon-Queue
iq.EnQueue (5);
rq.EnQueue (4.9);
```

- Bildung heterogener Kollektionen möglich, da Objekte auch Subklassen des Komponententyps angehören können

Entwicklung von Abstraktionsmechanismen in Programmiersprachen



Zusammenfassung

- Hauptziele objekt-orientierter Software-Entwicklung
  - bessere Erweiterbarkeit
  - bessere Wiederverwendbarkeit
- OO = ADTs (Klassen) + Vererbung + Polymorphismus
- Objekte und Klassen
  - statische vs. dynamische Objekte
  - Realisierung komplexer Objekte mit Objekt-Referenzen -> Aggregationshierarchien
  - Werte- vs. Referenzsemantik
  - vollständige Kapselung des Objektzustands vs. kontrollierte Zugriffsmöglichkeiten
- Vererbung von Attributen, Schnittstellen und Implementierung
  - Grundlage für einfache Erweiterbarkeit und Wiederverwendbarkeit
  - Substituierbarkeitsprinzip (Is-A-Hierarchie)
  - einfache vs. mehrfache Vererbung
- Polymorphismus
  - besonders wichtig in Form des Overriding
  - verlangt dynamisches Binden von Methoden