Objektorientierte Programmierung

Überblick

❏ Motivation für das objektorientierte Paradigma
  – Software-Design Methoden

❏ Entstehung des Objektmodells

❏ Objektorientierte Programmierung
  – Grundbegriffe
  – Basiskonzepte des objektorientierten Paradigmas

Literatur

ABC+83 M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshot and R. Morrison, “An Approach to Persistent Programming”, *The Computer Journal*, Vol. 26, No. 4, pp. 360-365, 1983.

Boo94 Grady Booch, *Object-Oriented Analysis and Design (with Applications)*, Benjamin/Cummings, Redwood (CA), 1994.

Cox86 Brad J. Cox. *Object Oriented Programming*. Addison Wesley, 1986.

CW85 Luca Cardelli, Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, Vol. 17, No. 4, Dec. 1985.

MaM88 Ole Lehrmann Madsen, Birger Møller-Pedersen, "What object-oriented programming may be — an what it does not have to be", *ECOOP ’88 – European Conference on OO Programming*, pp. 1 - 20, S. Gjessing, K. Nygaard [Eds.]; Springer Verlag, Oslo, Norway, Aug. 1988.

Mey86 Bertrand Meyer, "Genericity versus Inheritance", *Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA ’87*, pp. 391 - 405, Portland (Oreg., USA), published as *SIGPLAN Notices*, Vol. 21, No. 11, Nov. 1986.

Mey88 Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall Inc., Hemel Hempstead, Hertfordshire, 1988.

Str91 Bjarne Stroustrup. *The C++ programming language*, 2. ed., Addison-Wesley, 1991.

Str93 Bjarne Stroustrup, "A History of C++", *ACM SIGPLAN Notices*, Vol. 28, No. 3, pp.271 - 297, Mar. 1993.

Weg87 Peter Wegner, “Dimensions of Object-Based Language Design”, *OOPSLA ’87 – Conference Proceedings,* pp. 1-6, San Diego (CA, USA), published as *SIGPLAN Notices*, Vol. 22, No. 12, Dec. 1987.

Weg90 Peter Wegner, “Concepts and Paradigms of Object-Oriented Programming”, *ACM OOPS Messenger*, No. 1, pp. 8-84, Jul. 1990.

Newsgroup comp.object: FAQ (frequently asked questions) — sehr guter Überblick über Begriffe, Methoden, Sprachen, Produkte, etc.!

Motivation für das objektorientierte Paradigma

Ziele

❏ Mithalten mit der steigenden Komplexität großer Softwaresystemen

Booch [Boo94] spricht von *industrial-strength Software* und meint damit komplexe Softwaresysteme, wie sie heute in der Industrie (Prozeßsteuerung, Bank- und Versicherungswesen, Buchungssysteme, etc.) in großem Umfang eingesetzt sind. Solche Softwaresysteme sind so komplex, daß es in der Regel für einen Entwickler nicht möglich ist, alle Details des Designs zu überblicken. Die Softwaresysteme haben eine sehr lange Lebensdauer, haben viele Nutzer und werden von vielen unterschiedlichen Personen gewartet und erweitert.

Mit der ständig steigenden Leistungsfähigkeit (Geschwindigkeit, Speicherumfang, Parallelverarbeitung) von Rechnern nimmt auch die Komplexität der damit bearbeitbaren Probleme zu. Herkömmliche Softwareentwicklungsmethoden reichen zur Bewältigung dieser Komplexität nicht mehr aus. Objektorientierung ist eine neue Art Software zu entwickeln und zu strukturieren.

→ Softwarekrise: Hardware wird immer leistungsfähiger, Software wird größer, die aufgrund der Konkurrenzsituation zur Verfügung stehenden Entwicklungszeiträume werden gleichzeitig kürzer, die Kosten für Wartung und Weiterentwicklung steigen dramatisch.

❏ Produktivität des Programmierers steigern
  – bessere Erweiterbarkeit von Software
    Durch Modularisierung und klare Definition von Schnittstellen zwischen den einzelnen Softwarekomponenten
  – Wiederverwendung existierender Software(teile)
    Durch mehrfache Instantiierung allgemein einsetzbarer Softwaremodule (Modulbibliotheken) sowie durch Anpassung existierender Module an erweiterte oder abgewandelte Anforderungen (Vererbung).
  – bessere Kontrolle über Komplexität und Kosten von Software-Wartung
    Modulgrenzen und -schnittstellen legen klare Grenzen fest.

Software-Design Methoden (aus [Boo94])

Grundprinzip Dekomposition:
Zerlegen eines komplexen Problems in überschaubare Teilprobleme

❏ Top-down structured design (composite design)
  • Traditionelle Software-Design-Methode

❏ Data-driven design
  • Orientiert sich an der Abbildung von Eingabedaten auf Ausgabedaten

❏ Object-oriented design
  • "Moderne" Methode - findet seit einigen Jahren großen Zulauf
  • Bisherigen Erfahrungen zeigen vor allem Vorteile bei dem Design sehr großer Softwaresysteme

Software-Design Methoden

Top-down structured design (composite design)

➜ wesentlich durch die traditionellen Programmiersprachen (wie Fortran oder Cobol) beeinflußt

❏ Einheit für Dekomposition: **Unterprogramm**
- Resultierendes Programm hat die Form eines Baums, in dem Unterprogramme ihre Aufgaben durch Aufrufe weiterer Unterprogramme erledigen

❏ **Algorithmische Dekomposition** zur Zerlegung größerer Probleme
- Zentrale Sicht: Ausführung einer Problemlösung
  - Modularisierung: Identifizieren von Teilproblemen / Ausführungsschritten
  - Reihenfolge, in der Aktionen (=Teilproblemlösungen) auszuführen sind, wird hervorgehoben

❏ Eignung für die Strukturierung heutiger, sehr großer Softwaresysteme wird bezweifelt
- Heutige Rechner können Softwaresysteme bewältigen, deren Umfang und Komplexität die der 60er und 70er um Größenordnungen übersteigt
- Der Wert strukturierter Programmierung besteht nach wie vor, aber *Structured programming appears to fall apart when applications exceed 100 000 lines of code or so* [Boo94]
- Wesentliche Ursache ist wohl, daß nicht die zu bearbeitenden Daten, sondern die auszuführenden Aktionen im Mittelpunkt der Strukturierung stehen. Je größer ein Softwaresystem ist, desto größer wird die Datenmenge und die Menge der Prozeduren, die darauf arbeiten. Eine feste Zuordnung zwischen überschaubaren Teilmengen der Daten und Prozeduren bereits beim Softwaredesign wird durch algorithmische Dekomposition alleine aber nicht erreicht.

❏ *Strutured design* erfaßt nicht:
- Datenabstraktion & *Information Hiding*
  - Teilproblemlösungen operieren auf einem globalen Datenbestand — lokale Daten werden vor allem zur Zwischenspeicherung verwendet
- Nebenläufigkeit

❏ Probleme bei sehr komplexen Aufgabenstellung und objektbasierten oder objektorientierten Programmiersprachen
- *Structured design* resultiert in einer Problemstrukturierung, die nicht auf die Mechanismen in objektbasierten oder objektorientierten Sprachen abgestimmt ist

❏ Bislang am häufigsten eingesetzte Design-Methode
- Grundlagen von *structured design* beruhen auf den Arbeiten von Wirth, Dahl, Dijkstra und Hoare

Software-Design Methoden (2)

Data-driven design

❏ Grundlegende Arbeiten u. a. Jackson

❏ Softwarestruktur beruht auf Abbildung von Eingabe in Ausgabe
➜ Anwendung vor allem im Bereich *Information Management*

❏ Probleme vor allem mit zeitkritischen Ereignissen
- Zeitabläufe sind nicht erfaßbar!

Object-oriented design

Bertrand Meyer: *Rechner führen Operationen auf bestimmten Objekten aus; um flexiblere und wiederverwendbare Systeme zu erhalten, ist es daher sinnvoller, die Software-Struktur an diesen Objekten statt an den Operationen zu orientieren.*

siehe auch [Mey88]

Software-Design Methoden (3)

... Object-oriented design

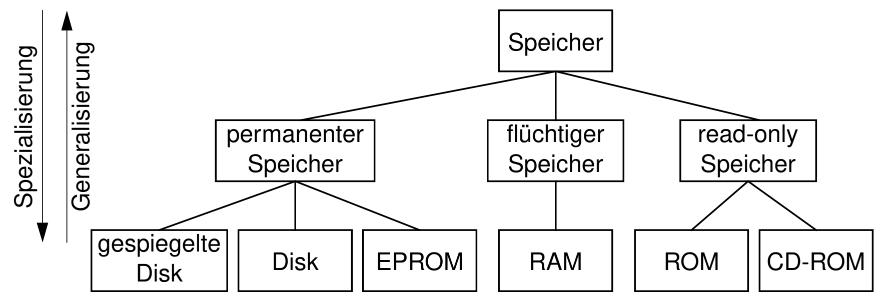
❏ Softwaresystem wird als Sammlung kooperierender Objekte modelliert

- Es stehen nicht auszuführende Aktionen im Mittelpunkt, sondern Einheiten, die einen Zustand besitzen, ihre Dienste anderen Einheiten anbieten und zur Erledigung ihrer Aufgaben andere Einheiten in Anspruch nehmen.

❏ einzelne Objekte sind Instanz einer Klasse in einer Hierarchie von Klassen

- Eine Klasse beschreibt die Eigenschaften von Objekten = Instanzen dieser Klasse
- Hierarchie entsteht durch die Beschreibung von Ober- und Unterklassen
  - Oberklasse: beschreibt allgemeine Eigenschaften von Objekten
  - Unterklasse: beschreibt Spezialisierung gegenüber der Oberklasse
    Instanzen der Unterklasse haben die durch die Oberklasse beschriebenen Eigenschaften, ergänzt oder abgeändert durch die in der Unterklasse beschriebene Spezialisierung

❏ Beispiel einer Klassenhierarchie:



- Die Oberklasse **Speicher** beschreibt die allgemeinen Eigenschaften eines Speichers
- Die Unterklassen **permanenter Speicher**, **flüchtiger Speicher** und **read-only Speicher** spezialisieren diese Eigenschaften — ein permanenter Speicher hat z. B. die Eigenschaft, daß sein Inhalt über einen Systemstart hinweg erhalten bleibt, während ein flüchtiger Speicher dies nicht garantiert. Ein read-only-Speicher würde bei einer Schreiboperation einen Fehler melden.
- **Gespiegelte Disk**, **Disk** und **EPROM** sind ihrerseits Unterklassen der (relativ hierzu) Oberklasse **permaneter Speicher**

Software-Design Methoden (4)

... Object-oriented design

❏ in der Struktur modernerer, höherer Programmiersprachen reflektiert

- Smalltalk
- C++
- Java
- Ada

❏ Grundlage: objektorientierte Dekomposition

- Dekomposition, orientiert an den Einheiten des *Problem domain* — nicht an den Einheiten oder Ausführungsschritten der Problemlösung (wie bei algorith. Dekomposition)

➔ essentieller Unterschied zu algorithmischer Dekomposition und damit dem *Top-down structured design*
  - andere Art, über Dekomposition nachzudenken
  - Software-Architektur wird völlig anders

➔ Software-Designer muß den *Problem-domain* verstehen und überblicken — bei traditioneller Software-Entwicklung kommt diese (eigentlich selbstverständliche) Voraussetzung häufig zu kurz.

❏ Vorteile:

\+ Wiederverwendung gemeinsamer Mechanismen

- Oberklassen beschreiben gemeinsame Eigenschaften, die in Unterklassen übernommen werden können.

➔ Software wird kleiner

\+ Software leichter zu ändern und weiterzuentwickeln

- Änderungen sind immer örtlich begrenzt (auf Klasse und evtl. deren Unterklassen)
- keine globalen Datenstrukturen
- Wiederverwendung durch gemeinsame Oberklassen garantiert gemeinsame Eigenschaften
  - Änderung in der Oberklasse wirkt sich konsistent auf alle Unterklassen aus
  - keine Fehler bei Reimplementierungen
- Weiterentwicklung durch Bildung von Unterklassen

\+ Ergebnisse weniger komplex

- Inkrementelle Entwicklung aus kleinen, überschaubaren Systemen
- OOD legt die Aufteilung großer Zustandsräume nahe (*separation of concerns*)

Entstehung des Objektmodells

Generationen von Programmiersprachen ([Boo94])

- ❑ Erste Generation (1954 - 1958)
  - Mathematische Ausdrücke (FORTRAN I, ALGOL 58)
    - Zielrichtung: Anwendungen im technischen und wissenschaftlichen Bereich
    - Umsetzung von mathematischen Problemen in Maschinensprache
    - erster Schritt in Richtung "Problem-Bereich", weg von der Maschine
- ❑ Zweite Generation (1959 - 1961)
  - Unterprogramme, getrennte Übersetzung (FORTRAN II)
  - Blockstruktur, Datentypen (ALGOL 60)
  - Datenbeschreibung, Dateibehandlung (COBOL)
  - Listenverarbeitung, Zeiger (Lisp)
    - Schwerpunkt dieser Generation: Algorithmische Abstraktion
    - Rechner wurden immer leistungsfähiger, Ziel war es, zu beschreiben "was der Rechner tun soll" (wie z. B. erst Datensätze lesen, dann sortieren, zuletzt ausdrucken)
    - weiterer Schritt näher an den "Problem-Bereich", weg von der Maschine
- ❑ Dritte Generation (1962 - 1970)
  - Zusammenfassung verschiedener Konzepte (PL/I)
  - verschiedene ALGOL-Nachfolger (ALOGOL 68, Pascal)
  - Klassen, Datenabstraktion (Simula)
    - Leistungsfähigkeit der Rechenanlagen war bis Ende der 60er Jahre weiter gestiegen (vor allem durch Transistortechnik und aufkommende integrierte Schaltkreise), Kosten waren gleichzeitig stark gesunken.
    - entsprechend größere Probleme wurden damit bearbeitbar
    - Programmiersprachen wurden im Hinblick auf Datenabstraktion weiterentwickelt (Algol, Pascal)
    - weiterer Schritt näher an den "Problem-Bereich"
- ❑ *The Generation Gap* (1970 - 1980)
  - viele neue Programmiersprachen, nur wenige haben sich verbreitet
    - Zielrichtung in aller Regel die Unterstützung größerer, komplexerer Software
    - Dabei entwickelte Konzepte wurden häufig in erfolgreiche Nachfolger früherer Sprachen übernommen. Es entstanden u. a.
      - Smalltalk (als Nachfolger von Simula!)
      - Ada (aus Algol68 und Pascal, mit Einflüssen aus Simula, Alphard und CLU)
      - CLOS (aus Lisp, LLOPS und Flavours)
      - C++ (aus C und Simula)
      - Eiffel (aus Simula und Ada)
    - in dieser Phase haben sich auch die objektbasierten und objektorientierten Sprachen herausgebildet

Objektorientierte Programmierung

Definition (Grady Booch [Boo94])

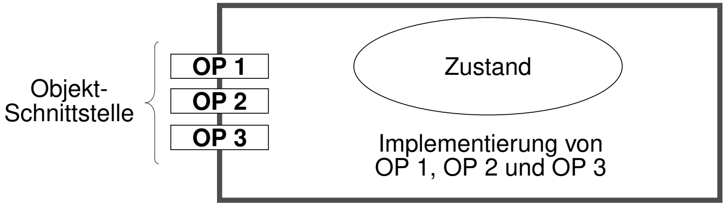
OOP ist eine Methode der Implementierung, in der Programme in Form von
Mengen kooperierender Objekte
organisiert sind, wobei jedes Objekt
Instanz einer Klasse
ist und die Klassen Bestandteil einer über
Vererbungsbeziehungen
definierten Hierarchie von Klassen sind.

- Diese Definition enthält die wesentlichen Grundbegriffe objektorientierter Programmierung:

- ❑ Grundbegriffe
  - ➜ Objekt
    - Objektorientierte Programmierung verwendet Objekte und nicht Algorithmen als die grundlegenden Bausteine der Problemlösung
  - ➜ Klasse
    - Jedes Objekt ist Instanz einer Klasse
  - ➜ Vererbung
    - Klassen stehen über Vererbungsbeziehungen in Beziehung miteinander

Objekte & Methoden

❑ **Objekt** = Variablen (Zustand) + Operationen



- Zusammenfassung von Datenstrukturen und Operationen auf diesen Datenstrukturen zu einer Programmiereinheit
- Klassisches Beispiel: Ein Stack, dessen Zustand als verkettete Liste implementiert ist und der die Operationen *push* und *pop* anbietet
- Vorteile des Objekt-Konzepts:
  – Logisch zusammengehörige Programmteile stehen auch im Programmtext beieinander
  – Das Ergebnis von Operationen hängt nicht nur von den Aufrufparametern (wie bei Funktionen) ab, sondern auch vom Zustand des Objekts (→ Operationen mit "Gedächtnis")
  – Teile des Objekts können vor Zugriff von "außen" geschützt werden — der interne Aufbau des Objekts kann "geheim" bleiben (**Information Hiding**)

❑ **Methoden** = Operationen eines Objekts (Schnittstelle)

- Die Methoden eines Objekts definieren gleichzeitig das **Verhalten** des Objekts

➜ **Objektbasierte Programmiersprachen**

- Eine Programmiersprachen ist objekt-basiert, wenn sie Objekte als Sprachkonstrukt unterstützt [Weg87]

- Probleme:
  – Benötigt man von einem Objekt mehrere Exemplare (z. B. mehrere Stacks, um unterschiedliche Informationen abzulegen), muß das Objekt entsprechend oft implementiert werden

  → Abhilfe: Schablone zur Erzeugung von Objekten

Klassen

❑ **Klasse** = Objektbeschreibung

- Eine Klasse beschreibt die Implementierung der Operationen von Objekten der Klasse und deren Zustandsstruktur (welche Variablen die Objekte enthalten)
- Man findet in der Literatur auch die "umgekehrte" Definition [Booch]:
  Eine Klasse ist eine Menge von Objekten gleicher Struktur und gleichen Verhaltens

❑ **Instanz** = aus einer Klasse erzeugtes Objekt

❑ **Instantiierung** = Erzeugung eines Objekts einer Klasse

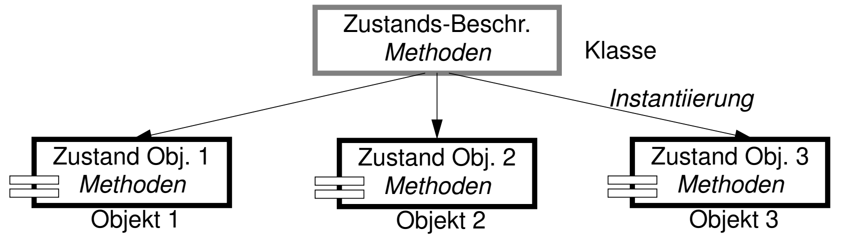
- Alle Instanzen können den in der Klasse implementierten Code für die Methoden gemeinsam nutzen, haben aber jeweils eine eigene Version des Objektzustands (der Variablen), gemäß der Strukturbeschreibung in der Klasse.

❑ **Instanzvariablen** = Variablen eines Objekts (= Zustand)

➜ **Klassenbasierte Programmiersprachen**
= Objekte & Klassen

Definitionen:

- Eine Programmiersprachen ist klassen-basiert, wenn jedes Objekt eine Klasse besitzt [Weg87]
- Programmiersprachen, die die Formulierung von Klassen und die Instantiierung von Objekten aus Klassen erlauben



Objekte und Klassen in C++

❑ Klassendeklaration vergleichbar mit C-Strukturdeklaration
- Unterschiede:
  - Klasse enthält Daten und Funktionen (=Methoden)
  - Daten der Klasse teilweise nur für Funktionen der Klasse sichtbar

❑ Zugriff auf Klassenkomponenten wie auf Strukurkomponenten mit den Operatoren `.` bzw. `->`

❑ Beispiel:

```
// Class counter
class counter
{
  private:
    int value;
  public:
    void incr() { value++; }
    void decr() { value--; }
    int get_value() { return value; }
};
```

❑ Syntax:

```
class Klassenname {
  private:
    Datendeklarationen und Methodendefinitionen
    (Daten und Methoden sind nur durch Methoden der Klasse
     zugreifbar)
  protected:
    Datendeklarationen und Methodendefinitionen
    (Daten und Methoden sind nur durch Methoden der Klasse
     und ihrer Unterklassen zugreifbar - vgl. Abschnitt zu
Vererbung)
  public:
    Datendeklarationen und Methodendefinitionen
    (Daten und Methoden sind aus allen Funktionen und
     Methoden des Programms zugreifbar)
};
```


Methoden in C++

- Methoden sind in ihrem Aufbau normale Funktionen (wie in C).
- Unterschied:
  - Eine Methode hat Zugriff auf geschützte Daten ihrer Klasse, während andere Funktionen und Methoden anderer Klassen diese Daten nicht erreichen können.

❑ Definition innerhalb der Klassendeklaration

– Methode wird als ***inline***-Funktion behandelt
- *inline*-Funktionen werden zur Übersetzungszeit an den Aufrufstellen expandiert
  - Code wird mehrfach erzeugt → Programm wird größer
  - Funktionsaufruf-Code (Register retten, etc.) wird eingespart → Programmausführung wird schneller
- Das Resultat einer *inline*-Expansion ist mit der Makro-Expansion von C vergleichbar. Die Semantik eines Aufrufs einer *inline*-Methode oder -Funktion ist aber im Gegensatz zur Makroexpansion identisch mit einem echten Funktionsaufruf (die Parameterübergabesemantik bleibt erhalten und Typprüfungen werden vom Übersetzer vorgenommen).
- vor allem für kleine und häufig genutzte Methoden sinnvoll

❑ Definition außerhalb der Klassendeklaration
- Innerhalb der Klassendeklaration wird nur der Funktionsprototyp angebenen (d. h. Funktionsname und Parameter-Typen).
- Die Methodendefinition erfolgt außerhalb der Klassendeklaration analog zu einer normalen Funktionsdefinition

– Klassenzuordnung über ***scope***-Operator `::`

– Methodenaufrufe wie normale Funktionsaufrufe
- Methoden, die außerhalb der Klassendeklaration definiert werden, werden beim Aufruf wie normale Funktionen behandelt — es findet keine *inline*-Expansion statt.

❑ Beispiel:

```
class counter {
  private:
    int value;
  public:
    void incr(); void decr(); int get_value();
};
void counter::incr()  { value++; }
void counter::decr()  { value--; }
int counter::get_value(){ return value; }
```

Instantiierung in C++

- Instantiierung von Objekten erfolgt
  - statisch zur Übersetzungszeit
    - Übersetzer reserviert Speicherplatz
  - dynamisch zur Laufzeit
    - Speicherplatz wird dynamisch vom Betriebssystem angefordert

statische Instantiierung

- durch Objektdefinition
  - Eine Klassendeklaration beschreibt dem Übersetzer nur den Aufbau der Klasse sowie den Code der Methoden — es wird (wie bei jeder C-Deklaration) aber kein Speicherplatz reserviert.
  - Eine Objektdefinition ist mit einer normalen Variablendefinition vergleichbar.
- Beispiel:

```
void main()
{
  counter c1;  // Objekt c1 der Klasse counter
  counter *pc1;// Zeiger auf ein counter-Objekt
  ...
}
```


Instantiierung in C++ (2)

dynamische Instantiierung

- C++ -Operatoren `new` und `delete`
  - entsprechen den Funktionen malloc( ) und free( )
  - sind auf beliebige Datentypen — damit auch auf Objekte (oder auch Felder) — anwendbar
  - liefern Zeiger auf eine Instanz des angegeben Datentyps (kein Cast erforderlich)
- Beispiele:

```
class counter
{ ... }
void main()
{
  counter c1;  // Objekt c1 statisch anlegen
  counter *pc1;// Zeiger auf ein counter-Objekt
...
  pc1 = new counter;
  pc1->incr();
  c1.incr();
  ...
  delete pc1;
  ...
}
```

  - Zeiger auf Objekte verhalten sich genauso wie Zeiger in C
  - **new** und **delete** können auch verwendet werden, um andere Daten zu allokieren.
  - Beispiele:

```
...
char *array;
int *ip;
...
// Feld für eine Zeichenkette der Länge num_chars anfordern
array = new char[num_chars + 1];
ip = new int;  // nicht gerade sinnvoll, aber korrekt
...
delete [] array;// Feld wieder freigeben
delete ip;      // einzelnen Integer wieder freigeben
...
```

Instantiierung in C++ (3)

Konstruktoren

Funktionen zum Initialisieren von Instanzen

❑ Konstruktor: Methode mit Methodenname = Klassenname
  – wird bei Instantiierung automatisch aufgerufen
    - statische Instantiierung:
      bei Programmstart bzw. Eintritt in den Block, in dem das Objekt definiert ist
    - dynamische Instantiierung:
      bei **new**-Aufruf
    - Parameterübergabe möglich

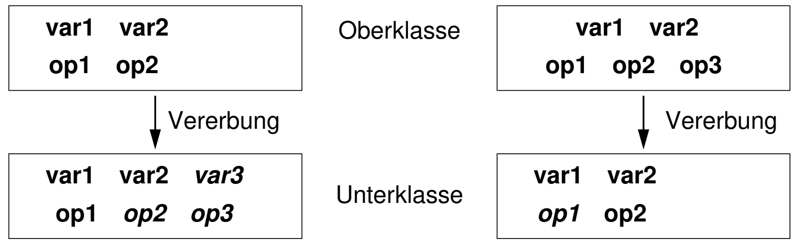
❑ Beispiel:

```
class counter {
  private:
    int value;
  public:
    counter(int c){ value = c; }// Konstruktor
    void incr(){ value++; }
   ...
};
...
counter c1(20);    // c1 instantiieren, value mit 20 initialisieren
cp = new counter(30);
```

- Im Rahmen einer Konstruktor-Funktion können beispielsweise:
  – Instanzvariablen initialisiert werden
  – Dynamisch mit dem **new**-Operator weitere Objekte oder Datenstrukturen (z. B. Felder variabler Länge, abhängig von einem Konstruktor-Parameter) instantiiert werden (der damit belegte Speicherplatz muß aber ggf. in der Destruktor-Funktion wieder freigegeben werden!!!).
- Instanzvariablen von Objekten werden nicht automatisch initialisiert!
  – Um eine *default*-Initialisierung von Instanzvariablen vorzunehmen, kann ein Standardkonstruktur ohne Parameter (Parameterliste enthält nur das Schlüsselwort **void**) definiert werden, der bei allen Instantiierungen ohne Parameter automatisch aufgerufen wird.

Vererbung

❑ **Vererbung** = Konstruktion von Klassen aus existierenden Klassen
  ➔ **Oberklasse** (Basisklasse) & **Unterklasse**



- Vererbung (engl. ***Inheritance***) ist eine Technik, die es erlaubt, neue Klassen auf bereits existierenden Klassen aufzubauen, statt sie vollständig neu zu programmieren. Die neu entstehende Klassen wird als **Unterklasse** (***Subclass, derived class***) bezeichnet, die Klasse, auf der aufgebaut wird, heißt **Oberklasse** (**Basisklasse**, ***Superclass***).
- Im Rahmen der Vererbung erbt die Unterklasse die Methoden und Variablen der Oberklasse. Außerdem kann die Unterklasse Variablen und Methoden hinzufügen, umdefinieren oder auch weglassen.

- Im linken Beispiel erbt die Unterklasse die Variablen var1 und var2 von der Oberklasse und definiert zusätzlich eine Variable var3. Die Implementierung der Methode op1 wird unverändert von der Oberklasse übernommen, die Methode op2 ist in der Unterklasse anders als in der Oberklasse implementiert, die Methode op3 kommt neu hinzu.
- Im rechten Beispiel wird die Zustandsbeschreibung (die Variablen) der Oberklasse für die Implementierung der Unterklasse unverändert übernommen. Die Methode op2 wird ebenfalls übernommen, die Methode op1 wird neu für die Unterklasse implementiert. Die Methode op3 der Oberklasse wird nicht übernommen (das Weglassen von Methoden der Oberklassen ist allerdings nicht in allen objektorientierten Sprachen möglich!).

❑ Vorteile:
  – Die Implementierung der Oberklasse wird nicht verändert
  – Gleiche Funktionalität muß nicht mehrfach implementiert werden
  – Veränderungen der Oberklasse wirken auf alle Unterklassen
  – Beziehungen zwischen Klassen werden dokumentiert

Vererbung in C++

- ❑ Unterklasse erbt Variablen und Methoden von der Basis-Klasse
- ❑ Unterklasse kann Basisklasse modifizieren
  - zusätzliche Methoden und Variablen
  - veränderte Methoden
- ❑ Methoden der Unterklasse haben Zugriff auf *public*- und *protected* -Bereich der Basisklasse
  - public-Basisklasse
    - ➔ die *Schnittstelle* der Basisklasse wird vererbt
      - Daten und Methoden der Kategorien *public* und *protected* haben diese Zugriffseigenschaften auch in der Unterklasse, d. h.
        - auf *public*-Daten und -Methoden von Instanzen der Unterklasse, die aus der Basisklasse übernommen wurden, kann aus Funktionen und Methoden anderer Klassen zugegriffen werden
          - ➔ die Unterklasse stellt einen **Subtyp** des Typs der Basisklasse dar (vgl. Abschnitt über Typisierung)!
        - wird diese Unterklasse im Rahmen weiterer Vererbung wieder als Basisklasse genutzt, können Methoden der weiteren Unterklasse ("Unter-Unterklasse") auf *public*- und *protected*-Daten und -Methoden der ursprünglichen Basisklasse zugreifen, soweit diese in der ersten Unterklasse nicht überlagert wurden.
  - private-Basisklasse
    - ➔ die *Schnittstelle* der Basisklasse wird *nicht* vererbt
      - Daten und Methoden der Kategorien *public* und *protected* werden mit der Zugriffseigenschaft *private* in die Unterklasse übernommen, d. h.
        - *public*-Daten und -Methoden der Basisklasse sind bei Instanzen der Unterklasse *private* und damit nicht außerhalb sichtbar.
        - Dadurch sind alle Methoden der Basisklasse an Instanzen der Unterklasse nicht aufrufbar, es sei denn, sie wurden explizit für die Unterklasse neu definiert (eine solche Methode der Unterklasse hat dabei auch die Möglichkeit, auf die entsprechende Methode gleichen Names der Basis-Klasse zuzugreifen, da für sie die *private*-Einschränkung ja nicht gilt).
          - ➔ der Typ der Unterklasse ist mit dem Typ der Basisklasse nicht in jedem Fall verträglich, ist also **kein Subtyp** (vgl. Abschnitt über Typisierung)!
        - *public*- und *protected*-Daten und -Methoden der Basisklasse sind für Unterklassen der Unterklasse nicht sichtbar.
- ❑ *private*-Daten und -Methoden der Basisklasse nicht sichtbar

Vererbung in C++ — Beispiel (1)


```
// Class counter
class counter
{
  protected:
    int value;
  public:
    void incr() { value++; }
    void decr() { value--; }
    int get_value() { return value; }
};
// Subclass resettable counter
class rcounter : public counter
{
  private:
    int initial;
  public:
    rcounter(int v){ initial = v; value = v; }
    void reset()   { value = initial; }
};
```

Zugriff auf Variablen der Basisklasse durch Methoden der Unterklasse

- Die Klasse **`rcounter`** erbt alle Methoden und Variablen von der Klasse **`counter`**.
- Die Variable **`value`** der Basisklasse ist als **`protected`** gekennzeichnet, um Unterklassen Zugriff zu ermöglichen.
- **`rcounter`** besitzt zusätzlich einen Konstruktor, der die Variable **`value`** der Basisklasse initialisiert.
- **`rcounter`** besitzt außerdem eine zusätzliche Methode **`reset()`**, die ein Rücksetzen des Werts auf den initialen Zustand ermöglicht.
- Inkrementieren, Dekrementieren und Abfrage von **`rcounter`**-Objekten erfolgt über die ererbten Methoden **`incr()`**, **`decr()`** und **`get_value()`**.
- Die Unterklasse erbt *public* von der Basisklasse, damit die Methoden **`incr()`**, **`decr()`** und **`get_value()`** für Instanzen der Unterklasse den Status *public* behalten.

Vererbung in C++ — Beispiel (2)


```
// Class window
class window
{
  protected:
    int x, y, width, height;
  public:
    virtual void init(int x, int y, int w, int h) { initialisieren
    virtual void move(int x, int y) { bewegen }
    virtual void display() { anzeigen }
    virtual void delete() { löschen }
};
// Subclass bordered window
class bordered_window: public window
{
  public:
    virtual void display() { gerahmtes Fenster anzeigen }
    virtual void change_width(int x) { Breite verändern }
    virtual void change_hight(int y) { Höhe verändern }
};
```

- Die Unterklasse *bordered_window* übernimmt die Instanzvariablen und die Methoden *init*, *move* und *delete* von der Basisklasse *window*.
- Die Methode *display* wird neu implementiert.

Referenzierung der Oberklasse aus der Unterklasse:

- Es ist prinzipiell möglich, aus der Methode *display* der Unterklasse auch die Methode *display* der Oberklasse aufzurufen. Zur Referenzierung von Methoden der Oberklasse (*Superclass*) wird häufig ein spezielles Schlüsselwort — z. B. *super* — verwendet. In C++ wird dies durch den Namen der Oberklasse und den Scope-Operator :: erreicht. In dem Beispiel könnte somit im Rahmen der *display*-Methode zuerst der Rahmen gezeichnet werden und anschließend der Objektzustand über einen Aufruf *window::display( )* durch die Methode der Oberklasse ausgeben werden.

Referenzierung der Unterklasse aus anderen Methoden der Unterklasse oder Oberklasse

- Zur Referenzierung anderer Methoden der gleichen Klasse wird normalerweise ein spezielles Schlüsselwort — *self* oder *here*— verwendet; in C++ gibt es zu diesem Zweck den Zeiger *this*. Soll z. B. im Rahmen der Implementierung der Methode *change_width* die Methode *move* aufgerufen werden, könnte dieser Aufruf durch *this->move(X,Y)* erfolgen (wobei in diesem Zusammenhang die verwendung von *this->* redundant ist).
- Bei Vererbung bezieht sich die Referenz *this* für die Unterklasse immer auf die Unterklasse — auch wenn *this* in einer Methode benutzt wird, die von der Oberklasse geerbt wurde. Erfolgt im Rahmen der Methode *move* ein Aufruf *this->display( )*, wird die Methode *display* der Unterklasse angesprochen, obwohl die Methode *move* in der Oberklasse implementiert wurde und nur über Vererbung in die Unterklasse übernommen wird. (In C++ müssen Methoden als *virtual* deklariert werden, damit diese Eigenschaft bei Vererbung auch tatsächlich funktioniert — sonst keine *richtige* Vererbung!)

Basiskonzepte des OO Paradigmas

Eine Reihe programmiersprachlicher Konzepte, die unabhängig von den Ideen zur objektorientierten Programmierung entstanden sind, sind für das objektorientierte Paradigma von grundlegender Bedeutung:

- ❑ Abstraktion
  - Kapselung
  - Abstrakter Datentyp
- ❑ Modularisierung
- ❑ Hierarchie

Weitere allgemeine programmiersprachliche Konzepte ergänzen viele objektorientierte Programmierumgebungen. Teilweise erhielten solche Konzepte auch erst durch die Verbindung mit dem objektorientierten Paradigma praktische Bedeutung.

- ❑ Typisierung
  - Typhierarchie
  - Polymorphismus
  - Generizität
- ❑ Nebenläufigkeit
- ❑ Persistenz

Abstraktion

Grundlegendes Konzept zur Lösung komplexer Probleme

❑ Für Zusammenhänge relevante Aspekte hervorheben

❑ Details vernachlässigen

- Abstraktion hilft dem Software-Entwickler, sich auf das zu lösende Problem zu konzentrieren und sich den Blick auf die eigentlichen Probleme nicht durch Implementierungsdetails zu verstellen!

➜ Objektorientierung

❑ wichtig:

- Signatur eines Objekts
  - Beschreibung der Objektschnittstellen — Methoden, Parameter, Ergebnisse
- Semantik eines Objekts
  - Auswirkung von Methodenaufrufen
- Eine Klasse realisiert die Implementierung einer Abstraktion. Für den Anwender dieser Klasse — bzw. ihrer Instanzen — ist dabei lediglich der für ihn sichtbare Teil interessant.
  - Sichtbar sind zum einen die Schnittstellen (=Methoden, Parameter und Resulate), zum anderen die Auswirkungen von Operationen auf das zukünftige Verhalten des Objekts.
  - Änderungen des Objektzustands, die über die Schnittstellen nicht beobachtet werden können (z. B. ein Stack-Objekt fordert dynamisch Speicher nach, um weitere Daten speichern zu können und gibt ihn auch selbsttätig wieder frei, wenn er nicht mehr benötigt wird), sind für den Benutzer des Objekts irrelevant.

❑ unwichtig:

- Implementierung eines Objekts
  - Instanzvariablen
  - Implementierung von Methoden
- Die Implementierung eines Objekts sollte ausgewechselt werden können, solange sich die Schnittstelle und die Semantik der Operationen nicht ändert. Dafür ist es aber wichtig, daß der Benutzer keine Kenntnisse über Implementierungsdetails erhält, die für seine Anforderungen nicht erforderlich sind. Er könnte sonst Annahmen über das Verhalten des Objekts machen und sich auf diese Annahmen bei seiner Programmierung verlassen, obwohl solch ein Verhalten durch den Implementierer des Objekts nie zugesichert wurde.

❑ zuerst die Abstraktion beschreiben, dann Implementierung vornehmen

- = zuerst Problemlösung spezifizieren, dann implementieren!

Kapselung

= Information Hiding

Verbergen der Implementierung einer Abstraktion vor den Benutzern der Abstraktion

❑ Gegenstück zu Abstraktion

- Abstraktion stellt die äußeren Eigenschaften eines Objekts heraus
- Kapselung verbirgt die Interna

❑ Grundvoraussetzung für Abstraktion

B. Liskov *Damit Abstraktionen funktionieren, müssen Implementierungen gekapselt sein*

❑ Kapselung in der Objektorientierung

- Darstellung des Objektzustands
- Implementierung der Methoden

➜ Abstrakter Datentyp

Abstrakter Datentyp

- ❑ Umsetzung der Konzepte *Abstraktion* und *Kapselung*
- ❑ Modell zur Beschreibung von
  - – Eigenschaften von Daten
  - – Operationen auf diesen Daten & deren Auswirkungen
  - – NICHT: Darstellung der Daten
- ❑ Spezifikation → Beschreibung aller wesentlichen Eigenschaften
- ❑ Überspezifikation: Angaben über Repräsentation des ADT

  - Das Konzept des ADT ist schon relativ alt — auch für Sprachen wie C oder Pascal gibt es Anleitungen zur Implementierung von ADTs mit Hilfe von globalen Daten und Funktionen, die auf diesen Daten operieren. Diese Sprachen bieten dem Programmierer aber keine Unterstützung zur Einhaltung der Kapselung der Daten.
  - Modula-2 und Ada unterstützen ADTs durch die Möglichkeit, ADT-Definition und -Implementierung separat zu beschreiben.

➔ Objektorientierung
- ❑ Klasse = Implementierung eines ADT
- ❑ Datenabstraktion: Objektzustand nur über Methoden erreichbar

Abstrakter Datentyp (2)

- ❑ Beispiel: Spezifikation eines ADT "Stack" [Mey88]

```
TYPES
        STACK[X]

FUNCTIONS
        empty: STACK[X] → BOOLEAN
        new: → STACK[X]
        push: X × STACK[X] → STACK[X]
        pop: STACK[X] ⇸ STACK[X]
        top: STACK[X] ⇸ X

PRECONDITIONS
        pre pop(s: STACK[X]) = (not empty(s))
        pre top(s: STACK[X]) = (not empty(s))

AXIOMS
        ∀x:X, s:STACK[X]:
        empty(new())
        not empty(push(x,s))
        top(push(x,s)) = x
        pop(push(x,s)) = s
```

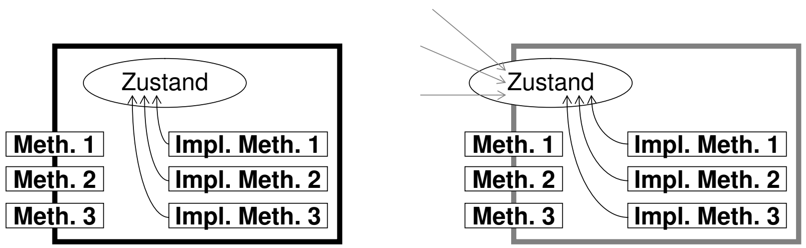
Die ersten beiden Abschnitte der Datentyp-Spezifikation beschreiben die Syntax des Datentyps, die anderen beiden seine Semantik:

- TYPES
  Gibt den zu spezifizierenden Datentyp (oder auch mehrere) an. In dem Beispiel ist das der Datentyp STACK mit einem Parameter X, durch den der Typ der Daten, die in dem Stack abgelegt werden können, festgelegt wird (STACK ist damit ein parametrisierbarer oder generischer Datentyp — siehe auch den Abschnitt "Generizität").
- FUNCTIONS
  beschreiben die Signatur der Dienste, die Instanzen des Datentyps anbieten:
  $f: E_1 \times E_2 \times \ldots \times E_n \rightarrow A_1 \times A_2 \times \ldots \times A_m$
  die Funktion f hat n Eingabeparameter der Typen E_1 bis E_n und liefert m Ergebnisse der Typen A_1 bis A_m.
  Partielle Funktionen (⇸ statt →) können nur unter bestimmten Vorbedingungen ausgeführt werden!
- PRECONDITIONS
  beschreiben die Vorbedingungen für partielle Funktionen. Im Beispiel werden die Funktionen pop und top für einen leeren Stack verboten.
- AXIOMS
  legen die Semantik des Datentyps fest.

Abstrakter Datentyp (3)

❏ Datenabstraktion in OOP

➜ auf Objektzustand kann nur durch die Methoden des Objekts zugegriffen werden



Objekt als Implementierung eines ADT — Objekt ohne Datenabstraktion

- Viele Programmiersprachen erzwingen Datenabstraktion allerdings nicht (z. B. C++)
- Häufig ist ein Kompromiß zwischen klarer Strukturierung und Disziplin sowie Flexibilität und Effizienz erforderlich.
- Vor allem in der Betriebsprogrammierung ist Datenabstraktion in der Praxis nicht bis in die untersten Betriebssystemschichten durchzuhalten.
  – Hinter einem Objekt kann sich "ein Stück Hardware" (z. B. ein Controller oder ein Prozessor) verbergen — Änderungen am Zustand solcher Objekte sind in manchen Fällen trotzdem von "außen" zu beobachten (z. B. weil nach einem Aufruf an ein Objekt, das den Treiber für eine serielle Schnittstelle realisiert, ein Signalpegel auf der Schnittstelle verändert wird und dadurch eine Telefonmodem-Verbindung abgebaut wird).

❏ Datenabstraktion in C++

- Sichtbarkeitsbereiche (*Scope*-Regeln)
  - Kapselung durch *private*- und *protected*-Bereiche in Klassen
  - Objektschnittstellen im *public-Bereich* einer Klasse

  - *private* -Daten und -Methoden
    – nur innerhalb von Methoden der gleichen Klasse sichtbar
  - *protected* -Daten und -Methoden
    – nur für Methoden der gleichen Klasse und in Methoden von Unterklassen sichtbar
  - *public* -Daten und -Methoden
    – für alle Funktionen des Programms und für alle Methoden anderer Klassen sichtbar

      public -Methoden bilden die Schnittstelle der Klasse nach außen

      public -Daten erlauben eine Verletzung der Objektkapselung
      ➜ vermeiden!

Modularisierung

Aufteilung eines Programms in einzelne Komponenten kann seine Komplexität reduzieren

❏ Teilproblem überschaubarer

❏ Teilprobleme Entwicklergruppen zuzuordnen

❏ Module = separate Übersetzungseinheiten

vor allem: Aufteilung erzeugt Schnittstellen

➜ Schnittstellen müssen klar definiert werden

➜ Schnittstellen müssen dokumentiert werden

❏ viele Programmiersprachen unterscheiden zwischen Schnittstelle und Implementierung eines Moduls

➜ enge Beziehung zwischen Modularität und Kapselung

- Die Unterstützung von Modularisierung in Programmiersprachen ist allerdings sehr unterschiedlich:
  – In C oder C++ sind Module lediglich getrennt übersetzbare Dateien — Schnittstellen zwischen Modulen können über Include-Dateien konsistent gehalten werden. Die Sprachen sehen allerdings keinerlei Mechanismen vor, die eine konsistente Benutzung der Schnittstellen wirklich garantieren.
  – Andere Sprachen (wie z. B. Modula) kontrollieren dagegen die Konsistenz der Modulschnittstellen direkt.

❏ *Structured Design*: Gruppierung von Unterprogrammen

- Unterprogramme werden meist aufgrund ihrer Interaktionsbeziehungen auf Module verteilt.

❏ OOD: Gruppierung von Objekten und Klassen

- Bei der Programmierung komplexer Softwaresysteme entsteht eine große Anzahl von Klassen
  ➜ viele kleine Einheiten mit eigenen Schnittstellen
  ➜ Beziehungen zwischen den Einheiten schwer überblickbar
- Aufgabe der Modularisierung im Rahmen von objektorientiertem Softwareentwurf ist damit, eine Gruppierung von Klassen auf Basis der Problemstruktur vorzunehmen. Diese kann sich von der Struktur, die durch die Interaktionsbeziehungen von Unterprogrammen entstehen würde stark unterscheiden!

Hierarchie

- ❑ Modularisierung & Abstraktionsbildung bei einer großen Problemstellung
  - ➔ große Zahl unterschiedlicher Abstraktionen
  - ➔ Überblick über Abstraktionen wird schwierig
- ❑ Zusammenhänge zwischen Abstraktionen
  - – Gemeinsamkeiten → allgemeinere Abstraktionen
  - – Unterschiede → Spezialisierung
  - ➔ Hierarchie: Ordnung auf Abstraktionen
- ❑ Objektorientierung
  - – Klassenstruktur: "**Art-von**"-Hierarchie (*kind of*)
    - Vererbung
    - Delegation
  - – Objektstruktur: "**Teil-von**"-Hierarchie (*part-of*)
    - Aggregation — ein Objekt ist aus mehreren Unterobjekten zusammengesetzt
      - – Beispiel: Objekt Auto besteht aus Unterobjekten Motor, Getriebe, Lenkung, etc.

Typisierung

- ❑ Typkonzept baut auf ADT-Theorie auf
  - Ein Typ ist eine genaue Charakterisierung der gemeinsamen Eigenschaften (bezüglich Struktur und Verhalten) einer Menge von Einheiten (Definition aus [Boo94]).
  - Bei Objekten wird der Typ durch die Signatur und die Semantik der Methodenaufrufe bestimmt.
  - Die meisten Programmiersprachen ermöglichen dem Programierer allerdings nur die Formulierung der syntaktischen Eigenschaften eines Typs.
    - – Bei objektorientierten Sprachen erfolgt dies häufig durch die Beschreibung einer Klassenschnittstelle.
      - ➔ Die Beschreibung einer Klasse definiert gleichzeitig einen Typ.
- ❑ Typisierung ermöglicht die Überprüfung von Ausdrücken auf Typ-Kompatibilität zur Übersetzungszeit
  - ➔ Vermeidung von Fehlern
    - Zuweisungen unterschiedlicher Typen oder die Übergabe falscher Typen als Parameter von Funktions- bzw. Methodenaufrufen kann zur Übersetzungszeit festgestellt werden. Auf diese Weise lassen sich viele Fehler von vorneherein vermeiden.
- ❑ Strenge Typisierung
  - – Kompatibilität aller Ausdrücke, die Werte repräsentieren, kann zur Übersetzungszeit überprüft werden
  - ➔ Einschränkung der Flexibilität
    - kompatible Typen
      - – Häufig kann der Compiler nicht feststellen, daß zwei unterschiedliche Typen kompatibel sind und damit in einem Ausdruck (z. B. einer Zuweisung) problemlos zusammen verwendbar sind.
    - dynamisches Binden
      - – Wenn erst zur Laufzeit festgelegt werden soll, welches Objekt (oder auch welche Funktion in nicht-OOP) über eine Variable erreichbar ist, kann die Typ-Verträglichkeit nicht generell zur Übersetzungszeit überprüft werden.
  - ➔ mehr Flexibilität durch Polymorphismus und Generizität
    - – Polymorphismus erlaubt neben der Verwendung des passenden Typs auch andere, konforme Typen.
    - – Generizität ermöglicht eine Parametrierung mit Typen. Dadurch können z. B. Klassen, die für verschiedene Typen einsetzbar wären, aufgrund strenger Typisierung aber nicht flexibel verwendet werden können, durch Typ-Parameter für die Verwendung in Zusammenhang mit einem Typ eingestellt werden.

Typhierarchie

- ❏ Häufig 1:1-Beziehung zwischen Klassen und Typen — aber nicht notwendig
  - – mehrere Klassen können einen Typ implementieren
    - Beispielsweise können unterschiedliche Klassen einen Datentyp "Stack" unterschiedlich implementieren (Daten werden in einem Feld oder in einer verketteten Liste verwaltet), aber die gleichen Methodenaufrufe sowohl bezüglich Signatur, als auch bezüglich Semantik anbieten.
  - – eine Klasse kann unterschiedliche Typen implementieren
    - Eine Klasse kann beispielsweise besondere Methodenaufrufe zur Einstellung von Parametern oder Debug-Möglichkeiten besitzen, deren Verwendung in Teilen einer Anwendungsimplementierung aber nicht erlaubt sein darf (weil die Klasse möglicherweise später durch eine Implementierung ohne diese Methoden ersetzt werden soll). Wenn man in den Teilen der Anwendung einen Typ für die Klasse bekanntgibt, der die besonderen Methodenaufrufe nicht enthält, garantiert das Typsystem, daß sie auch nicht verwendet werden. In den Teilen der Anwendung, die die besonderen Methodenaufrufe benötigen, wird ein anderer Typ für die Klasse deklariert, der die Methoden enthält.

- ❏ Hierarchie bei Klassen: Unterklassen ("**kind-of**"-Beziehung)
  - Ziel: Wiederverwendung
  - Unterklasse nicht immer typ-konform zu Oberklasse
    - In den meisten Programmiersprachen (auch in C++ und Eiffel) ist es möglich, Unterklassen zu programmieren, die nicht typ-konform zur Oberklasse sind.
    - Unter dem Gesichtspunkt, durch eine Vererbung bei Klassen Programmcode wiederzuverwenden ist dies eigentlich völlig unproblematisch. Da die meisten Programmiersprachen aber durch Klassendeklarationen gleichzeitig Typdeklarationen vornehmen, ist es wichig, unterscheiden zu können, wo durch Klassen-Vererbung ein konformer Untertyp entsteht und wo nicht.

- ❏ Hierarchie bei Typen: Untertypen ("**is-a**"-Beziehung)
  - Ziel: Beschreibung konformer Typen (➔ Polymorphismus)
    - Durch eine Typhierarchie soll ausschließlich festgelegt werden, welche Typen zu welchen anderen Typen konform sind.

- ❏ Typvererbung (*Subtyping*) als Mechanismus zur Ableitung von Typen
  - Analog zur Vererbung bei Klassen: der Untertyp erbt von einem Obertyp
  - Unterschied zur Vererbung bei Klassen: der Untertyp ist auf jeden Fall konform zum Obertyp — d. h. er kann zusätzliche Methoden enthalten, enthält aber bezüglich Signatur und Semantik alle Methoden des Obertyps.

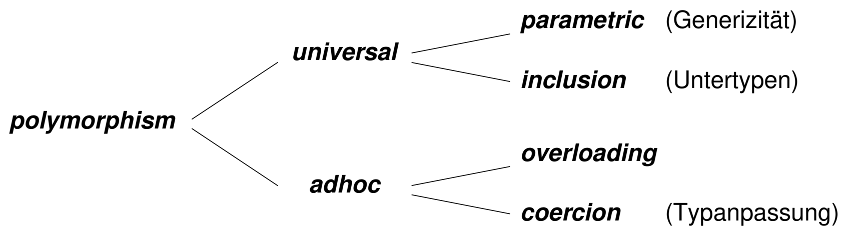
➔ Zusammenhänge zwischen Typen werden übersichtlich
  - – Identifikation konformer Typen
    - Durch die Typhierarchie ist festgelegt, welche (Unter-)Typen statt eines Typs in einem Ausdruck verwendet werden dürfen.

Polymorphismus

- ❏ *die Fähigkeit, verschiedene Formen anzunehmen* [Mey88]
  - – mehr als ein Typ für Werte oder Variablen
  - – verschiedene Typen als Parameter zu Funktionen
  - – verschiedene Typen als Operanden eines Operators

- ❏ Beispiel:
  +-Operator arbeitet mit int- und real-Werten

- ❏ Klassifikation [CW85]



universal polymorphism:
- funktioniert prinzipiell mit beliebig vielen Typen (alle Typen haben aber eine gemeinsame Grundstruktur — sind beispielsweise Subtypen eines gemeinsamen Obertyps, z. B. Zahlen in verschiedenen Darstellungen und Ausprägungen).

parametric polymorphism:
- – Eine polymorphe Funktion besitzt einen expliziten Parameter, in dem der Typ des Parameters für jeden Aufruf mit übergeben wird (spezielle Ausprägung dieses Konzepts in Adas *generic functions* zu finden).

***inclusion polymorphism**:*
- – Bei Subtyping: ein Untertyp kann jederzeit statt des Typs verwendet werden. **In objektorientierten Sprachen von großer Bedeutung**: häufig mit Vererbung gekoppelt — wenn eine Unterklasse einen Untertyp implementiert, können Instanzen der Unterklasse statt Instanzen der Basisklasse verwendet werden.

ad hoc polymorphism:
- funktioniert nur auf einer endlichen Menge — möglicherweise völlig voneinander unabhängiger Typen.

overloading:
- – Mehrere verschiedene Funktionen werden unter dem gleichen Namen definiert und es wird anhand des Aufruf-Kontexts (Anzahl und Typ der Aufrufparameter) entschieden, welche Implementierung aufgerufen wird. (Beispiele siehe C++)

coercion:
- – Vor dem Aufruf der Funktion werden Parameter, deren Typ nicht mit der Funktionsdefinition übereinstimmt, automatisch in den passenden Typ konvertiert. (Beispiel für *Coercion*: in C wird bei der Addition eines int- und eines real-Wertes automatisch eine Konvertierung beider Operanden nach double durchgeführt).

Polymorphismus in C++

Einige Konzepte in C++ können zur Realisierung von Polymorphismus eingesetzt werden. Dabei sind drei Kategorien zu unterscheiden:

- ❏ *overloading polymorphism*
  - – Function-name overloading
  - – Operator overloading
    - • Funktionen oder Operatoren eines Namens können für unterschiedliche Parameter- bzw. Operandentypen definiert werden.
- ❏ *inclusion polymorphism*
  - – public Vererbung & virtuelle Funktionen
    - • Da in C++ Subklassen gleichzeitig Subtypen des Datentyps der Basisklasse bilden, sind sie typkonform zur Basisklasse. Instanzen der Subklasse können daher statt Instanzen der Basisklasse verwendet werden. In der Subklasse können dabei Methoden der Basisklasse anders implementiert sein.
- ❏ *coercion polymorphism*
  - – Cast-Operatoren
    - • Für die Verwendung eines "unpassenden" Typs als Parameter oder in einem Ausdruck können für Einzelfälle Regeln beschrieben werden, wie der unpassende Typ in einen passenden Typ konvertiert wird. Diese Konvertierung erfolgt dann automatisch.

Function-name overloading

- ❏ mehrere Funktionen mit gleichem Namen innerhalb eines Gültigkeitsbereichs
- ❏ Übersetzer wählt die aufzurufende Funktion aus
  - ➔ statisches Binden (*early binding*)
    - • Kriterien: Parameterzahl & Parametertypen
- ❏ Beispiel:

```
int max (int a, int b) { return((a>b):a?b); }
float max (float a, float b) { return((a>b):a?b); }
char *max (char *a, char *b) {
        if ((strcmp(a,b)> 0) return a; else return b; }
```

- ❏ Spezialfälle:
  - – mehrere Konstruktoren für eine Klasse
  - – Operator-Overloading
  - – Überladen einer Methode bei Vererbung
    - • Function-name overloading kann auch Rahmen von Vererbung erfolgen, wenn in einer Subklasse eine Methode der Basisklasse mit anderer Signatur (Parameterzahl oder -typen bzw. Ergebnis) definiert wird.

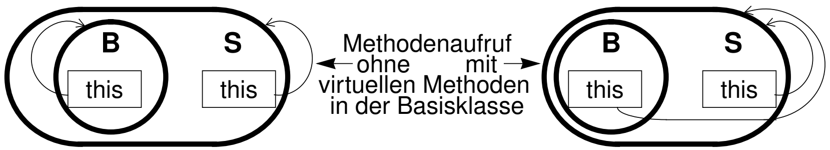
      In solch einem Fall wird vom Compiler anhand der Parametrierung der Methodenaufrufe festgestellt, ob die Methodenimplementierung der Basisklasse oder die der Subklasse aufgerufen wird. Die Methode gleichen Namens in der Subklasse ersetzt damit die Implementierung der Basisklasse nicht, sondern sie stellt eine andere, Aufruf-inkompatible Implementierung dar — muß also streng von *inclusion-Polymorphism* unterschieden werden (siehe Abschnitt über virtuelle Funktionen)!

virtuelle Funktionen (Methoden)

- ❏ Aufrufe "normaler" Methoden werden zur Übersetzungszeit gebunden
  - – bei Zeiger auf Objekt wird Methode durch Typ des Zeigers bestimmt
    - Wird über einen Zeiger auf ein Objekt einer Basisklasse eine Methode aufgerufen, so wird immer der in der Basisklasse definierte Code ausgeführt, unabhängig von dem Objekt, auf das der Zeiger zur Ausführungszeit gerade zeigt.
  - – Methode eines Subtyps (Unterklasse) wird nicht gefunden
    - Wird an einen Zeiger auf ein Objekt einer Basisklasse eine Instanz einer Unterklasse zugewiesen (Unterklasse = Subtyp, daher konform und damit Zuweisung möglich), so sind über diesen Zeiger nach wie vor nur die Methoden der Basisklasse aufrufbar. Auch wenn in der Unterklasse eine Methode der Basisklasse neu definiert wurde, wird nicht der Code der Unterklassenmethode, sondern der Code der Basisklassenmethode ausgeführt!

- ❏ Aufrufe virtueller Methoden werden dynamisch gebunden (*late binding*)
  - – bei Zeiger auf Objekt wird Methode durch Instanz bestimmt
    - Wird über einen Objektzeiger eine Methode aufgerufen, so wird die Methode des Objekts, auf das der Zeiger gerade zeigt, ausgeführt.
  - – Methoden von Unterklassen (Subtypen) werden gefunden
    - Wird an einen Zeiger auf ein Objekt einer Basisklasse eine Instanz einer Unterklasse zugewiesen, so sind nach wie vor nur die in der Basisklasse deklarierten Methoden aufrufbar
      - – in der Unterklasse neu hinzugekommene Methoden sind im Typ des Zeigers (= Typ der Basisklasse) nicht angegeben und daher nicht bekannt.
    - Wurde eine Methode der Basisklasse in der Unterklasse umdefiniert, so wird der in Unterklasse implementierte Code ausgeführt.

- ❏ ohne virtuelle Methoden keine "richtige" Vererbung
  - ➜ Selbstreferenz (*this*-Pointer) wird bei Vererbung nicht angepaßt



- Wird in einer Methode der Basisklasse eine nicht-virtuelle Methode aufgerufen, die in der Subklasse umdefiniert wurde (direkt oder über den Zeiger *this*), so wird trotzdem der in der Basisklasse definierte Code aufgerufen — auch wenn dies in einer Instanz der Subklasse geschieht!
- Der Aufruf von virtuellen Methoden aus Methoden der Basisklasse entspricht dem Aufruf von Methoden von "außen" — d. h. es wird zuerst in der Subklasse und dann in der Basisklasse danach gesucht.

virtuelle Funktionen — Beispiel


```
class geo_obj {            // allgem. Basisklasse
  public:
    virtual void draw();
};
class circle : public geo_obj {// Subklasse
  public:
    void draw();
}
class square : public geo_obj {// Subklasse
  public:
    void draw();
}
main () {
  geo_obj *ptr;
  ptr = new circle;
  ptr->draw();
...
```

- Der Aufruf **`ptr->draw()`** bewirkt den Aufruf der **`draw`**-Methode der Subklasse **`circle`**, obwohl **`ptr`** ein Zeiger auf ein Objekt der Klasse **`geo_obj`** ist!
- Wäre **`draw`** in **`geo_obj`** nicht als virtual deklariert worden, hätte der Aufruf **`ptr->draw()`** die Methode der Klasse **geo_obj** angesprochen, obwohl eine Instanz der Unterklasse **`circle`** an **`ptr`** gebunden war.
- Die **`draw`**-Methoden der Unterklasse und auch die **`draw`**-Methode der Oberklasse **`geo_obj`** müssen in obigem Beispiel natürlich noch außerhalb der Klassen definiert werden.
- Auf die Definition der **`draw`**-Methode der Oberklasse könnte auch verzichtet werden, **`geo_obj`** wäre dann eine abstrakte Klasse (siehe Abschnitt über abstrakte Klassen).
  - – es könnten dann allerdings keine Instanzen von **`geo_obj`** erzeugt werden, da die Klasse nicht vollständig definiert ist
  - – **`geo_obj`** wäre dann nur als Basisklasse im Rahmen von Vererbung zu gebrauchen — in obigem Beispiel durchaus sinnvoll, weil eine **`draw`**-Methode, die nicht näher beschriebene geometrische Objekte zeichnen kann, wohl ohnehin keinen Sinn macht.

Virtuelle Methoden einer Subklasse, die eine virtuelle Methode der Basiklasse neu implementieren, sind automatisch wieder virtuell (z. B. bei weiterer Vererbung).

➜ **`void draw();`** in der Klasse **`circle`** ist damit korrekt, es muß nicht **`virtual`** davor angegeben werden!

Cast-Operatoren

Coercion-Polymorphismus

Automatische Typkonvertierungen für Parameter oder Operanden sind für Basistypen in den Programmiersprachen meistens vorgesehen.

- z. B. in C: short → int → double ← float

C++ erlaubt die Beschreibung von Typkonvertierungen auch für benutzerdefinierte Datentypen im Rahmen einer Klassendefinition.

- ❑ Coercion-Polymorphismus durch die Beschreibung von Typkonvertierungen in Klassendefinitionen
- ❑ Cast-Operator (-Methode) definiert Umwandlung des Klassen-Typs in einen anderen Typ

```
class::operator type(void) { ... }
```

  - Für den durch die Klasse **`class`** definierten Datentyp wird in **`{ ... }`** beschrieben, wie eine Konvertierung in den Datentyp **`type`** zu erfolgen hat.
  - Für das Beispiel der Klasse **`counter`** (vgl. Abschnitt über Klassen und Typen) würde ein Konvertierungsoperator von **`counter`** nach **`long`** folgendermaßen aussehen:

```
counter::operator long(void)
  { return( (long) value); }
```

  - Das Beispiel stellt die Definition der Konvertierungsoperators außerhalb der Klassendefinition dar — sie kann natürlich auch innerhalb der Klassendefinition (dann ohne **`class::`**) erfolgen!

- ❑ Cast-Operator entspricht unärem Operator

Cast-Operatoren — Beispiel

```
class time
{
  private:
    int hour, min, sec;
  public:
    ...                           // Konstruktor

    operator long () {            // cast-Operator
      return(hour*3600 + min*60 + sec);
    }
};
main() {
  time now(1,10,2);
  long t;

  t = now + 10;            // ⇒ t == 4212
...
```

- wird ein Objekt des Datentyps **`time`** an einer Stelle benutzt, an der ein Datentyp **`long`** erwartet wird, so wird automatisch der cast-Operator **`long`** der Klasse **`time`** aufgerufen und dessen Rückgabewert eingesetzt.

Typen — abstrakte Klassen

❑ Basisklasse deklariert Methoden und deren Parameter, definiert sie aber nicht
  – *pure virtual function*
  ➔ Basisklasse beschreibt einen Typ

❑ Subklassen definieren unterschiedliche Implementierungen der Methoden
  ➔ jede Subklasse stellt eine Implementierung des Typs dar

❑ Basisklasse ist nicht instantiierbar

❑ Beispiel:

```
class geo_obj {            // abstrakte Klasse
  public:
    virtual void draw() = 0;// pure virtual function
};
class circle : public geo_obj {// Subklasse
  public:
    void draw() { ... }
}
```

- Die Klasse **geo_obj** beschreibt lediglich die Signatur von geometrischen Objekten, enthält aber keinerlei Implementierungen.
- Unterklassen von **geo_obj** stellen dann konkrete Implementierungen verschiedener geometrischer Objekte dar, die alle konform zu dem Datentyp **geo_obj** sind (und damit z. B. an einen Zeiger auf **geo_obj** zuweisbar sind).

- Grundsätzlich können auch einzelne (virtuelle) Methoden, die für alle Subklassen gleich sein sollen, in der abstrakten Basisklasse definiert werden und andere, die erst in den Subklassen implementiert werden sollen, als *pure virtual functions* angegeben werden. Die Basisklasse ist dann nach wie vor eine abstrakte Klasse, da sie nicht vollständig implementiert ist, sie entspricht aber nicht mehr einer reinen Typdefinition, sondern ist eine Mischung aus Typdefinition und -implementierung.
- Genauso können Subklassen von abstrakten Klassen auf die Definition einer *pure virtual function* verzichten (sie müssen deren Deklaration aber explizit wiederholen!). Solche Subklassen sind dann nach wie vor abstrakte Klassen und können nur im Rahmen von weiterer Vererbung eingesetzt werden.

Generizität (*Genericity*)

❑ Möglichkeit, den Typ von programmiersprachlichen Einheiten durch Parameter festzulegen
  Beispiele:
  - generische Funktionsargumente (ein Funktionsparameter bestimmt den Typ anderer Parameter)
  - generische (parametrierbare) Klassen

❑ OOP: generische Klassen

❑ Instantiierung (+Parametrierung) ➔ aktuelle Klasse
  aktuelle Klasse ➔ Instantiierung von Objekten

❑ Beispiel:
  - allgemeine Klasse Stack
    – int-Stack
    – real-Stack
    – string-Stack

❑ Generizität kann weitgehend mit Vererbung nachgebildet werden [Mey86]
  Aber: eigentlich werden dadurch die Begriffswelten von Typ und Klasse (wie so oft) vermischt: Generizität ist ein Konzept, um mehr Flexibilität im Bereich Typisierung zu erreichen, während Vererbung Klassenhierarchien erzeugt!

❑ realisiert in Ada, Eiffel und C++ (ab. V 3.0, *Templates*)

Generizität — Templates in C++

❑ Klassendefinition ohne Festlegung auf bestimmte Typen

- Vor allem wenn Klassen in Standardbibliotheken bereitgestellt werden sollen, gibt es viele Fälle, in denen von vorneherein nicht festgelegt werden kann, für welche Typen solch eine Klasse gebaut werden soll. Typische Beispiele sind Container-Klassen wie Listen, Vektoren oder assoziative Felder.

- Grundsätzliche gibt es zwei Möglichkeiten, die Konstruktion solcher Klassen in einer Sprache zu unterstützen:
- dynamische Typprüfung zur Laufzeit
  - Die betreffenden Typen werden zur Übersetzungszeit nicht festgelegt und durch dynamische Typprüfung zur Laufzeit wird eine konsistente Verwendung von Parametern und Instanzvariablen überprüft. Insbesondere in nicht-getypten Sprachen (wie Smalltalk) wird dieser Weg gewählt. Die Programmierung wird dadurch einfacher, vor allem sind alle Klassen sehr flexibel einsetzbar. Zur Laufzeit entsteht aber beträchtlicher Mehraufwand durch die dynamischen Typprüfungen.
- statische Typprüfung zur Übersetzungszeit + parametrierbare Klassen
  - Die korrekte Verwendung von Typen bei Parameterübergabe und bei Zuweisungen wird zur Übersetzungszeit komplett überprüft. Um Klassen definieren zu können, ohne sich auf alle in der Klassenimplementierung verwendeten Typen von vorneherein festlegen zu müssen, werden parametrierbare Klassen eingeführt: Die Klassendefinition enthält Parameter, die in der Implementierung statt eines Typs eingesetzt werden können.

❑ Template = parametrierbare Klasse

❑ Beispiel:

```
template <class T> class stack {
  private:
    int index;
    T *array;
  public:
    void stack(int n)
           { index = 0; array = new T[n]; }
    void push(T elem)
           { array[index++] = elem; }
    T pop(void)
           { return(array[index--]); }
};
```

- In der Klassendefinition können neben einem oder mehreren Typparametern auch andere Parameter auftreten. In dem Beispiel könnte man neben dem Typ **T** beispielsweise auch die Stackgröße als Template-Parameter angeben. Man könnte dann ein Objekt "Integer-Stack mit 10 Elementen" instantiieren. Dieses Objekt hätte aber einen anderen Typ als ein "Integer-Stack mit 20 Elementen". Wird die Größe dagegen im Konstruktor angegeben, so instantiiert man Objekte vom Typ "Integer-Stack" und übergibt dem Konstruktor die gewünschte Größe. Der Typ solcher Objekte ist dann identisch.

Generizität — Templates in C++ (2)

❑ Klassen-Parametrierung bei Objekterzeugung

- C++ sieht keinen separaten Schritt zur Parametrierung eines Templates vor: Parametrierung und Objektinstantiierung werden durch ein Sprachkonstrukt beschrieben.
  – Stroustrup begründet diese Vorgehensweise damit, daß nur, wenn ein Programm vollständig vorliegt, bekannt ist, welche Templates instantiiert werden müssen. Viele Templates werden in Bibliotheken bereitgestellt und Instantiierungen erfolgen direkt oder auch indirekt durch Benutzer, die möglicherweise nicht einmal etwas über die Existenz dieser Templates wissen. Eine explizite Instantiierung von parametrierten Klassen aus Templates wird daher nicht für sinnvoll gehalten [Str93].

❑ Beispiele:

- Instantiierung und Benutzung eines Float-Stacks:

```
stack <float> s(10);
float a;
s.push(a);
a = s.pop( );
```

- Instantiierung eines Integer-Stacks:

```
stack <int> s(10);
...
```

- Das Stack-Template ist in der beschriebenen Implementierung nicht problemlos für beliebige Typen (also beispielsweise Zeichenketten, komplexe Zahlen, etc.) verwendbar, da eine direkte Zuweisung an die Feldelemente erfolgt. Dies funktioniert nur, wenn der **=**-Operator für den jeweiligen Typ definiert ist. Flexibler könnte man mit Referenzen arbeiten, diese wurden hier aber nicht behandelt (für weitere Informationen siehe vor allem [Str91]).

Nebenläufigkeit (Concurrency)

Nebenläufigkeit: mehrere Aktivitätsträger werden parallel von mehreren Prozessoren bearbeitet oder von einem Prozessor simuliert

❑ Nebenläufigkeit orthogonal zu Objektorientierung
aber: Komplexität einer Problemlösung wird durch Nebenläufigkeit erhöht
zusätzliche Probleme:
- Koordinierung
  - gegenseitiger Ausschluß
- Synchronisation
  - warten auf Ergebnisse anderer Aktivitätsträger
- Verteilung
  - Verteilen der auszuführenden Objekte auf die verfügbaren Prozessoren

❑ Granularität: Nebenläufigkeit / Objekte (Kapseln)
- Nebenläufigkeit feiner granular
  ➜ NL auch objektintern
  - mehrere Aktivitätsträger gleichzeitig in einem Objekt
    → objektinterne Koordinierung erforderlich
- Nebenläufigkeit grober granular
  ➜ NL nur objektextern
  - immer nur maximal ein Aktivitätsträger zu jedem Zeitpunkt in einem Objekt
    → keine Koordinierungsprobleme innerhalb eines Objekts, Koordinierung kann auf die Objektinteraktionen beschränkt werden

❑ Integration der Kontrolle von Nebenläufigkeit in objektorientierte Sprachen
- orthogonale Sprachen
  - Orthogonale Sprachen enthalten keine Sprachkonstrukte zur Kontrolle von Nebenläufigkeit. Zur Koordinierung muß der Programmierer sprach-externe Mechanismen wie z. B. Semaphore einsetzen.
- nicht-orthogonale Sprachen
  - In nicht-orthogonalen Sprachen können Objekte gegen nebenläufige Ausführung geschützt werden. Dies wird entweder automatisch für alle Objekte vorgenommen
    ➜ **uniforme Sprachen**
    oder es kann durch Sprachkonstrukte vom Programmierer festgelegt werden, ob ein Objekt geschützt werden soll
    ➜ **nicht-uniforme Sprachen**.

Persistenz

Motivation für Persistenz [ABC+83]

❑ "aktive" Daten → Programmiersystem
- "Aktive" Daten, die *kurzfristig* im Speicher angelegt sind, werden mit den Hilfsmitteln des Programmiersystems (Sprache) bearbeitet.

❑ "passive" Daten → DBMS oder Dateisystem
- "Passive" Daten, die die Programmausführung überleben sollen, werden einer Datenbank (DBMS) oder einem Dateisystem übergeben und mit dessen Funktionen verwaltet.

➜ 2 Sichten auf Daten

➜ Nachteile für den Anwender
- Konvertierung notwendig
  - Programmieraufwand für die Konvertierung und den Transport zwischen passivem und aktivem Zustand (*scanf*/*printf*-Anweisungen)
    (Atkinson schätzt i. a. ca. 30% des Codes!)
- Datenschutz des Programmiermodells geht verloren
  - Schutzmechanismen des Programmiermodells (z. B. Typisierung) greifen nur, solange Daten aktiv sind - bei der ersten Wandlung kann alles verloren gehen.

allgemeine Definition

❑ Persistenz ist die Eigenschaft von Daten, das Ende einer Umgebung, in der sie entstanden oder benutzt wurden, zu überdauern

❑ verschiedene Stufen von Persistenz
1. Zwischenergebnisse von Ausdrücken
2. lokale Variablen
3. globale Variablen (überleben das Ende von Funktionsaufrufen)
4. Daten, die über mehrere Programmausführungen hinweg existieren
5. Daten, die zwischen verschiedene Versionen eines Programms existieren
6. Daten, die Programme überleben

➜ die Punkte (1) bis (3) werden überlicherweise durch Programmiersprachen erbracht, die Punkte (4) bis (6) durch Dateisysteme bzw. DBMS

Persistenz (2)

Persistenz in objektorientierten Systemen

- Objekte überleben das Ende der Umgebung (Aktivitätsträger, Anwendungsausführung), in der sie instantiiert oder benutzt wurden
- In OO-Betriebssystemen mächtiger Basis-Mechanismus
  - Datenhaltung
  - Datentransport
- Beispiele
  - Dateisysteme
    - Eine traditionelle Datei ist ein persistentes Objekt mit Methoden wie *read*, *write* und *seek*.
  - Datenbanksysteme
  - persistente Kommunikationsobjekte
    - Beispielsweise ein Puffer-Objekt
- Eigenschaften des objektorientierten Programmiermodells gehen automatisch auf Mechanismen über, die damit erstellt wurden
  Beispiele:
  - Vererbung zur Realisierung unterschiedlicher Dateiobjekte
  - Kontrolle von Nebenläufigkeit

OOA & OOD

Objektorientierte Analyse

Eine Analysemethode, die aus der Sicht der Klassen und Objekte, die im Vokabular des Problembereichs identifiziert werden können, den Bedarf untersucht.

Objektorientiertes Design

Eine Design-Methode, die den Prozeß der objektorientierten Dekomposition und eine Notation zur logischen und physischen Beschreibung von statischen und dynamischen Modellen des zu beschreibenden Systems umfaßt

- Vielzahl von Design- und Analyse-Methoden veröffentlicht
  Beispiele:
  OMT — *Object Modeling Technique* (Rumbaugh)
  James Rumbaugh. *Object-oriented modeling and design,* Prentice-Hall International, Englewood Cliffs, NJ, 1991 (inzwischen in Booch-Methode integriert)

  OOD — *Object Oriented Design* (Booch)
  [Boo94]

  Coad/Yourdan
  Peter Coad, Edward Yourdon. *Object-oriented analysis*, 2. ed., Prentice Hall, Englewood Cliffs, NJ, 1991

  Shlaer/Mellor
  Sally Shlear, Stephen J. Mellor, *Object-oriented systems analysis*, Prentice Hall, Englewood Cliffs, NJ, 1988.

- CASE-Tools für OOD
  Beispiele
  - ParadigmPlus
  - Rational ROSE
  - ObjectMaker
  - TeamWork
  - diese CASE-Tools unterstützen in der Regel mehrere OOD-Methoden (meistens zumindest Booch, Coad/Yourdan, Rumbaugh)