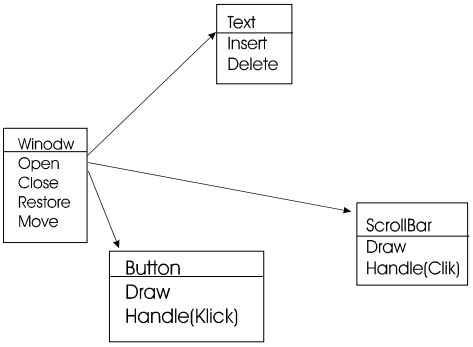
Objektorientierte Programmierung

WS 1995/1996

1 Einführung

1.1 Hauptideen von OOP

1.1.1 Menge kooperierender Objekte



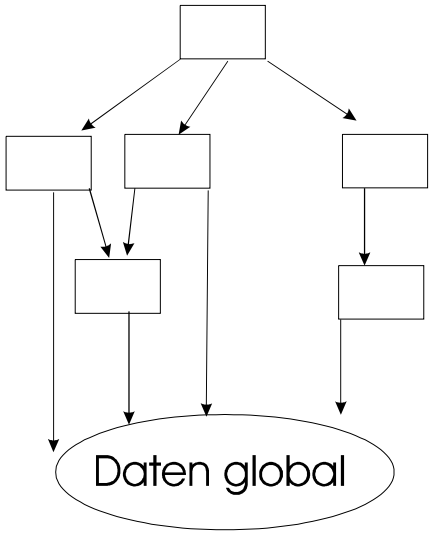
1.1.2 Arbeiten mit Varianten, ohne diese zu unterscheiden

1.1.2.1 Programmstruktur

1.1.2.1.1 Prozedural

Daten:

- Implementierungsdetails sichtbar
- unklar, welche Daten und Prozeduren zusammengehören
- kein Datenschutz



1.1.2.1.2 Objektorientiert

- jedes Objekt verwaltet eigenen Aufgabenbereich
- Implementierung der Daten versteckt
- Zusammengehörendes bildet eine Einheit
- Datenschutz

(Objekt) Klasse (Daten + Prozeduren)

1.1.2.2 Operationen

1.1.2.2.1 Prozedural

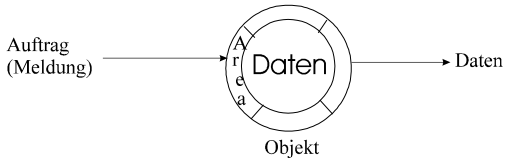
x := AreaOf (rectangle)

Problem: verschiedene Varianten von Daten

- AreaOfRectangle
- AreaOfCircle
- ...

⇒ **Fallunterscheidungen:** jedoch Probleme bei neuen Varianten

1.1.2.2.2 Objektorientiert



x := figure.Area ();

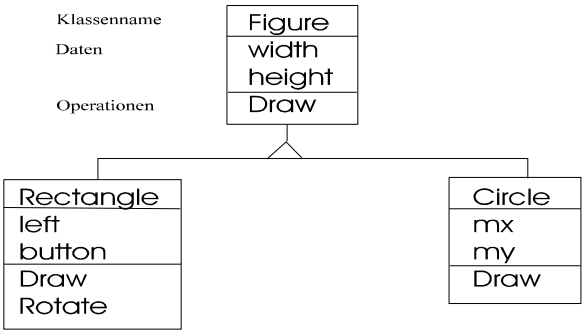
- figure kann verschiedenartige Objekte enthalten
- Objekt weiß selbst, wie Aufträge zu erledigen sind
- keine Fallunterscheidungen!
- neuer Datentype Triangle: *x := figure.Area ()* braucht nicht geändert werden

1.1.2.3 Merkmale objektorientierter Sprachen

1.1.2.3.1 Datenabstraktion

- InformationHiding
- Abstrakte Datentypen
- Baustein: Daten + Prozeduren
- Bausteine haben einen Zustand

1.1.2.3.2 Vererbung (Typerweiterung)



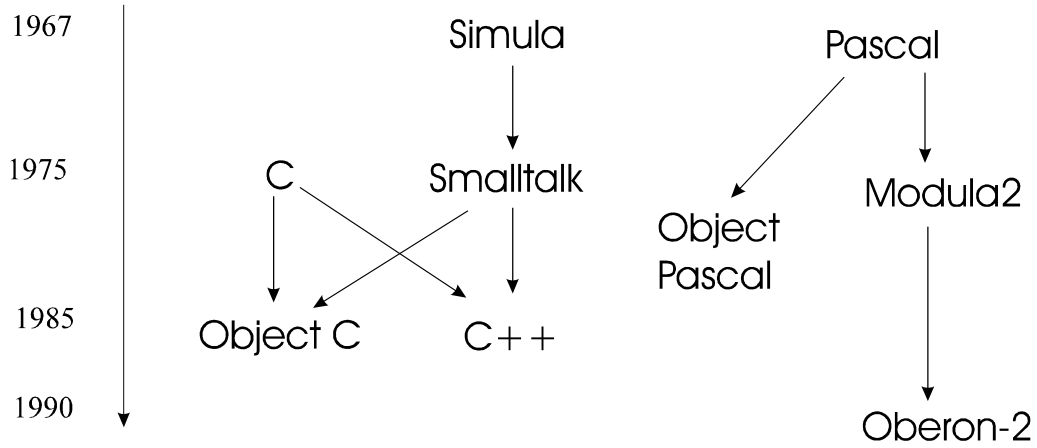
1.1.2.3.3 Dynamische Bindung (Polymorphismus)

Bindung von Aufträgen an Prozeduren

1.1.2.4 Terminologie

- Klasse: abstrakter Datentyp
- Objekt: Exemplar einer Klasse (Wert)
- Attribure, Instanzvariablen: Datenkomponente eines Objekts
- Methoden: Operationen eines Objekts
- Meldung: Auftrag (dynamisch gebundener Prozeduraufruf)
- Vererbung: Typerweiterung, Spezialisierung

1.1.2.5 Geschichte von Object-Oriented-Programming-Languages



weitere OOPL: Eiffel, Clos

2 Klassen

2.1 Implementierung in Oberon-2

```
TYPE    Counter = RECORD
                value : INTEGER;
        END;

PROCEDURE (VAR c : Counter) Add (x : INTEGER);
BEGIN
        c.value := c.value + x
END Add;

PROCEDURE (VAR c : Counter) Clear ();
BEGIN
        c.value := 0
END Clear;
```

Empfänger: (VAR c : Counter)

Eingangsparameter

- Objekt, auf das sich die Operation bezieht
- gibt Klasse an, zu der Operation gehört

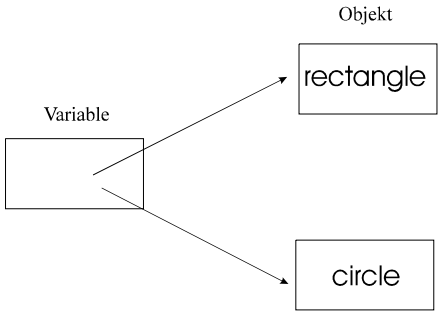
Unterschied zu prozeduraler Programmiersprache:
Procedure Add (VAR c : Counter; x : INTEGER);

2.1.1 Benutzung von Klassen

```
VAR     c1, c2 : Counter;
c1.Clear; c1.Add (3); c1.Add (5);
```

„Senden von Meldungen“

2.2 Objekte und Referenzen

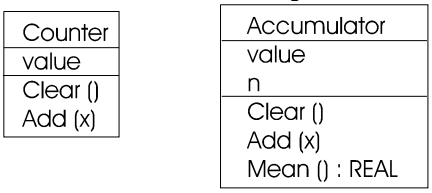


Klassen sind meist als Zeiger auf Record deklariert

```
TYPE    Counter = POINTER TO CounterDesc;
        CounterDesc = RECORD
                value : INTEGER;
        END;

PROCEDURE (c : Counter) Add (x : INTEGER); BEGIN ... END Add;
```

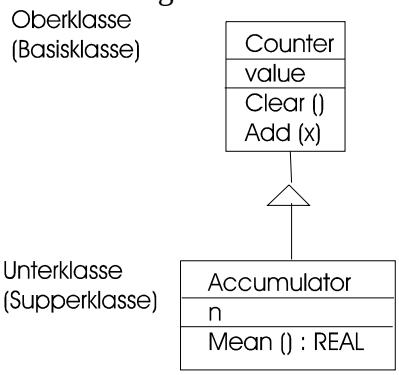
3 Vererbung

Problem:
vorhandene Bausteine passen nicht



Lösungsmöglichkeiten:

- neu schreiben
- Source kopieren und ändern
- Vererbung



3.1 Notation

```
TYPE    Accumulator = POINTER TO AccumulatorDesc;
        AccumulatorDesc = RECORD (CounterDesc)
                n : INTEGER
        END;

        PROCEDURE (a : Accumulator) Mean () : REAL;
        BEGIN
                RETURN a.Value/a.n
        END Mean;
```

3.2 Überschreiben geerbter Methoden (Overriding)

Parameterliste muß gleich bleiben!

```
PROCEDURE (a : Accumulator) Clear;
BEGIN
        a.n := 0;
        a.Clear↑
END Clear;

PROCEDURE (a : Accumulato) Add (x : INTEGER);
BEGIN
        a.Add↑ (x);
        INC (a.n)
END Add;
```

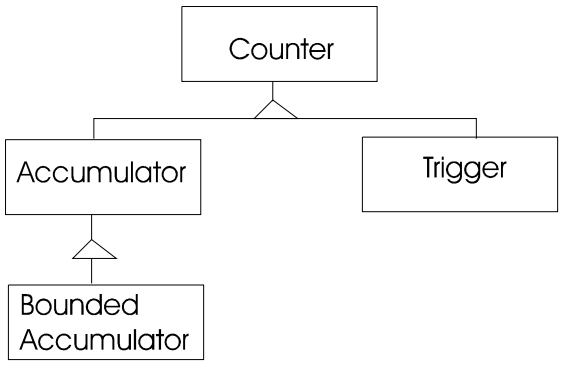
↑: Suppercall

3.3 Benutzung

VAR a: Accumulator;

a.Clear;
a.Add (3);
a.Add (4);
x := a.Mean ();

3.4 Klassenhierarchien



Baumdiagramm

- Vererbung ist transitiv: BoundedAccumulator ist Unterklasse von Counter, weil er Unterklasse von Accumulator ist, und Accumulator eine Unterklasse von Counter ist
- Ist-Beziehung: jeder Accumulator ist ein Counter. Nicht jeder Counter ist ein Accumulator (⇒ nicht reflexiv)
- Vererbung ist Spezialisierung

3.5 Typkompatibilität

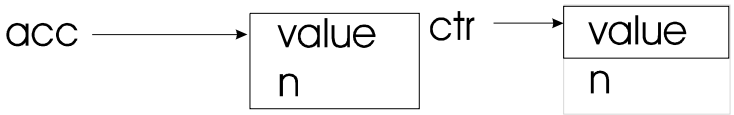
```
TYPE    Accumulator = POINTER TO AccumulatorDesc;
        AccumulatorDesc = RECORD
                value, n : INTEGER
        END;

        PROCEDURE (a : Accumulator) Clear; BEGIN ... END Clear;
        PROCEDURE (a : Accumulator) Add (x : ...); BEGIN ... END Add;
        PROCEDURE (a : Accumulator) Mean () : REAL; BEGIN ... END Mean;
```

Unterklasse ist kompatibel mit der Oberklasse

3.6 Zuweisung

```
VAR     ctr : Counter;
        acc : Accumulator;
        bounded : BoundedAccumulator;
        trigger : Trigger;
```

ctr := acc;



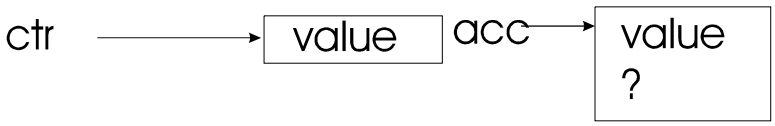
3.6.1 statischer Typ

Typ, mit dem die Variable deklariert ist

3.6.2 dynamischer Typ

Typ des Objekts, den die Variable zur Laufzeit enthält

~~*acc := ctr;*~~



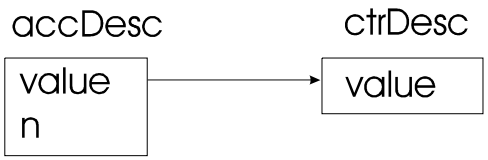
acc := bounded;
ctr := bounded;
~~*acc := trigger;*~~

```
PROCEDURE CountData (c : Counter);
BEGIN
        LOOP
                Read (x); ... c.Add (x);
                ...
        END;
END CountData;

CountData (ctr);
CountData (acc); x := acc.Mean ();
```

3.7 Recordzuweisungen

VAR ctrDesc : CounterDesc;
accDesc : AccumulatorDesc;

ctrDesc := accDesc;



Projektion

~~*accDesc := ctrDesc;*~~

```
PROCEDURE CountData (VAR c : CounterDesc);
        c.Add (x);
END CountData;
```

CountData (accDesc)

3.8 Variableninhalt

- Recordvariable vom Typ T enhält Objekte vom Typ T
- Var-Parameter-Record vom Typ T referenziert Objekte vom Typ T oder eine Erweiterung (Unterklasse)
- Zeiger-Variable vom Typ P = POINTER TO T referenziert Objekte vom Typ T oder eine Erweiterung (Unterklasse)

3.9 Typtest zur Laufzeit

ctr IS Accumulator ctr ≠ NIL

ist dynamischer Typ von Counter gleich Accumulator (oder eine Erweiterung davon)?

ctrDesc IS AccumulatorDesc

3.10 Typzusicherung zur Laufzeit (Type Guard)

ctr (Accumulator).n
wenn *ctr IS Accumulator*, dann Umwandlung, sonst Laufzeitfehler!

ctrDesc (AccumulatorDesc).n

- sichere Typumwandung, da vor Zuweisung getestet wird
- kostet Laufzeit

NEW (ctr); NEW (acc); NEW (bounded);
IF acc IS Accumulator ... TRUE
IF ctr IS Accumulator ... FALSE
IF acc IS Counter ... (immer) TRUE ⇒ von Sprache verboten

ctr := bounded;
IF ctr IS BoundedAccumulator ... TRUE
IF ctr IS Accumulator ... TRUE

acc := ctr (Accumulator);
ctr := acc;
~~*trigger := ctr (Trigger);*~~

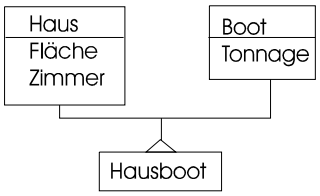
trigger := ctr (Trigger);

n := ctr (Accumulato).n;
x := ctr (Accumulato).Mean ();

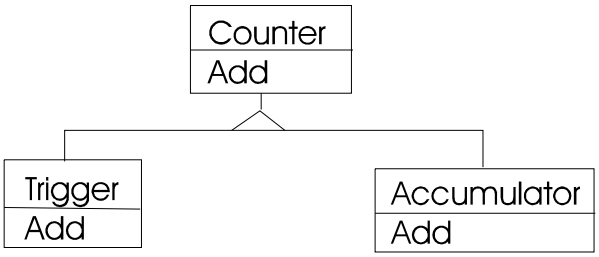
3.11 WITH-Anweisung

```
WITH ctr : Accumulator DO
        n := ctr.n;
        x := ctr.Mean ();
        |ctr : Trigger DO ...
        | ....
        ELSE ...
END;
```

3.12 Mehrfache Vererbung



4 Dynamische Bindung



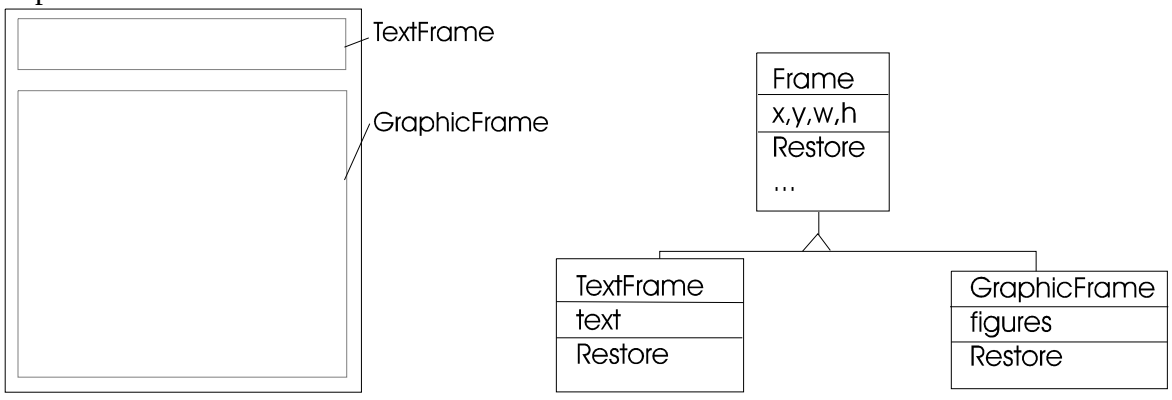

```
VAR     ctr : Counter;

NEW (ctr);
ctr.Add (3);                                    „CounterAdd“
ctr := acc;
ctr.Add (3);                                    „AccumulatorAdd“
```

Eine Meldung *object.M* führt zum Aufruf derjenigen M-Methode, die zum dynamischen Typ des Empfängerobjekts gehört.

Prozeduraufruf	**Compiler kennt Adresse von Add**
Add (ctr, 3)	„Call Add“ „statische Bindung“ (earlybinding)

Meldung	**Compiler kennt Adresse von Add nicht**
ctr.Add (3)	T := dynamischer Typ von ctr CALL Add von T „dynamische Bindung“ (latebinding)

Bsp.:



Restaurieren des Fensters:

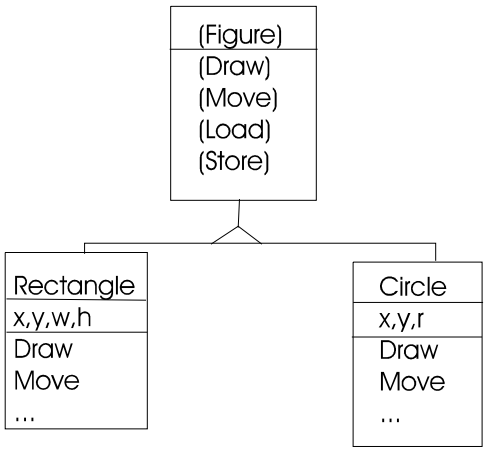
```
FOR allsubframes f DO
        f.Restore
END;
```

Vorteile:

- keine Fallunterscheidung
- neue Varianten sind leicht hinzuzufügen

4.1 Abstrakte Klassen

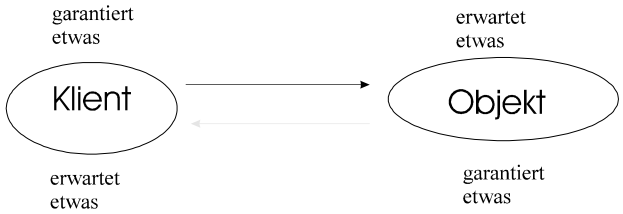
- definieren einer Schnittstelle
- i. a. keine Implementierung



Implementierung:

```
TYPE    Figure  = POINTER TO FigureDesc;
        FigureDesc = RECORD END;

PROCEDURE (f : Figure) Draw (...);
BEGIN
        HALT (99);      (* falls durch Supercall irrtümlich aufgerufen, erzeugt TRAP 99 *)
END Draw;
```

5 Kontrakte



{pre} obj.M {pos}

Wenn der Klient pre garantiert, garantiert M post.

5.1 Klassenassertionen

Für jede Methode

- **Precondition:** Prädikat über Eingangsparameter und Attribute
- **Postcondition:** Prädikat über Eingangs-/Ausgangsparameter und Attribute

Für jede Klasse

- **Invariante:** Prädikat über Attribute

Text
len buf
Insert (pos, str)

- **inv:** len ≥ 0
- **pre:** 0 ≤ pos ≤ len
- **post:** (len' = len + lenght (str)) & (buf[pos..pos + len - 1] = str)

Klasse korrekt, wenn gilt:

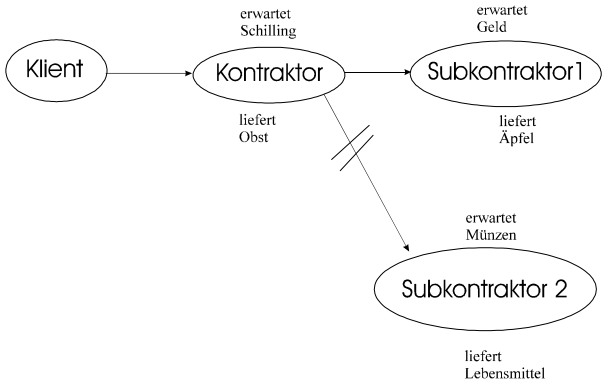
- {TRUE} Initialisierung {inv}
- jede Methode
    - inv unverändert läßt
    - bezüglich pre und post korrekt ist

5.2 Subkontrakte

Klasse kann Auftrag an Unterklasse (= Subkontraktor) weitergeben.

Subkontraktor darf

- nicht mehr erwarten
- nicht weniger garantieren



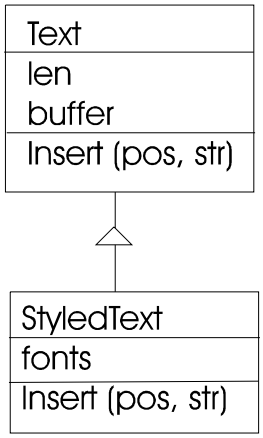
5.3 Assertionen in Unterklassen

$inv_{sub} = inv_{neu}$ & inv_{base}

$pre_{base} \Rightarrow pre_{sub}$ pre_{sub} darf schwächer sein

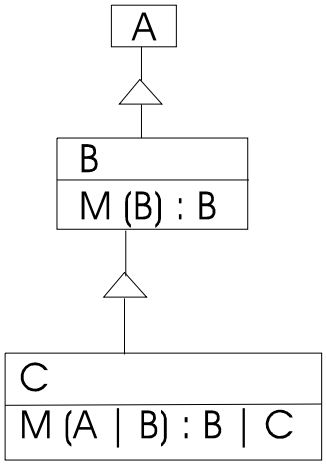
$post_{sub} \Rightarrow post_{base}$ $post_{sub}$ darf stärker sein

Bsp.:



- pos > 0: Font von str von buffer[pos - 1] übernehmen
- pos ≤ 0: Font von str = DefaultFont
- inv: (0 ≤ len) & (fonts[0..len - 1] = Font von buffer[0..len - 1])
- pre: -len ≤ pos ≤ len
- post: ((len' = len + length (str)) & (buf[|pos|..|pos| + len - 1])) &
  ((p ≤ 0) & (fonts[|pos| .. |pos| + len - 1] = DefaultFont) |
  ((p > 0) & (fonts[|pos| ..|pos| + len - 1] = fot[|pos| - 1]))

5.4 Parametertypen beim Überschreiben



Parameter von M in C?

- Eingangsparameter können vom Oberklassentyp sien (Kontravarianz)
- Ausgangsparameter können vom Unterklassentyp sein (Kovarianz)
- Übergangsparameter müssen vom gleichen Typ sein

6 Typische Anwedungen von OOP

6.1 Datenabstraktion

6.1.1 Andere Abstraktionsmittel als Klassen

6.2 Heterogene Datenstrukturen

6.2.1 Konventionelle Implementierung

6.2.2 Objektorientierte Implementierung

6.2.3 Erweiterbarkeit zur Laufzeit

6.2.4 Weitere Beispiele für heterogene Datenstrukturen

6.3 Generische Bausteine

6.3.1 Implementierung

6.3.2 Echte Generizität

6.4 Austauschbares Verhalten

6.5 Erweiterung und Anpassung von Bausteinen

7 Objektorientierter Entwurf

7.1 Prozedurorientierter Entwurf

7.2 Objektorientierter Entwurf

7.3 Allgemeine Vorgehensweise

7.3.1 Beispiel einer objektorientierten Zerlegung

7.4 Methode von Abbott

7.4.1 Probleme der Methode von Abbott

7.4.2 Aufgabenstellung: Bankomatsoftware

7.5 CRC-Karten (Class-Responsibilities-Collaborators)

7.6 Beziehugen zwischen Klassen

7.6.1 Collaboration-Graph

7.7 Häufige Enwurfsfehler

7.7.1 Verwechslung von Ist-Beziehung und hat-Beziehung

7.7.2 Verwechslung von Oberklasse und Unterklasse

7.7.3 Falsches Empfängerobjekt

7.8 Einige Tips

7.8.1 Wann sind Klassen sinnvoll

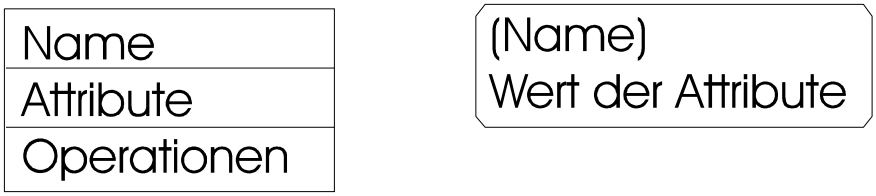
8 Object Modelling Technique (OMT)

(nach Rumbough)

Modelle = Abstraktion der Systeme,
- um zu verstehen
- um zu überprüfen
- um mit anderen zu kommunizieren

Abstrahieren
- Weglassen von Details
- Konzentration auf Wesentliches

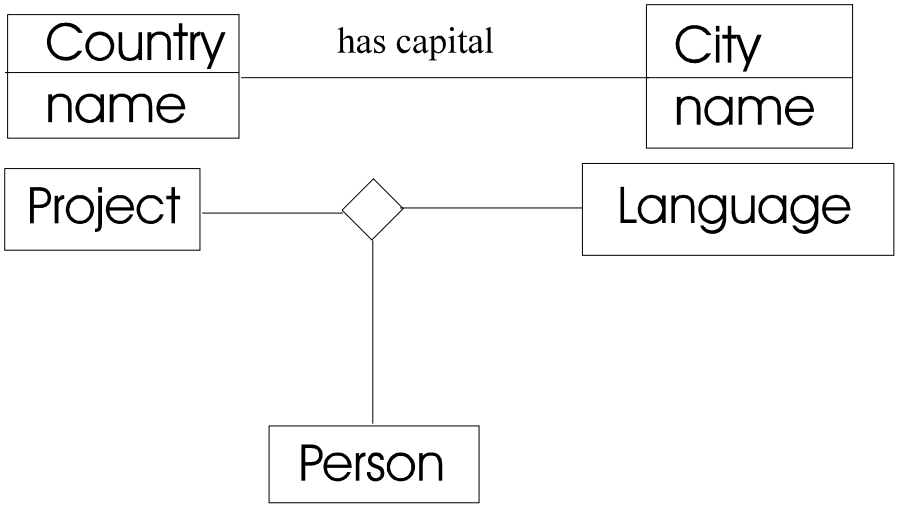
8.1 Objektmodell

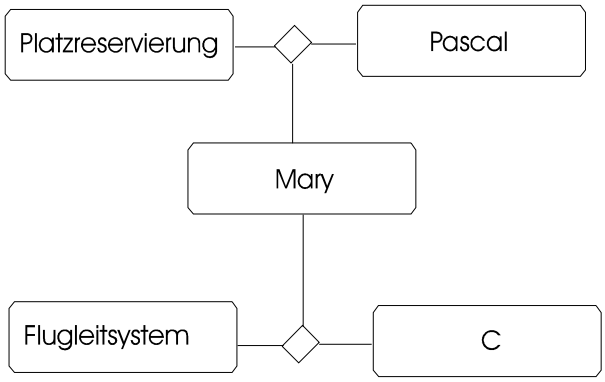


Klasse Objekt

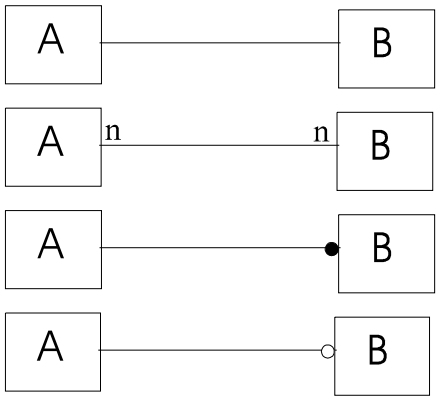
8.1.1 Beziehungen

8.1.1.1 Assoziationen (benutzt-Beziehung)





8.1.1.1.1 Anzahl der Beziehungen



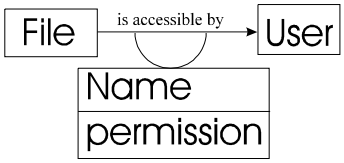
1 A benützt 1 B

n A benützen n B

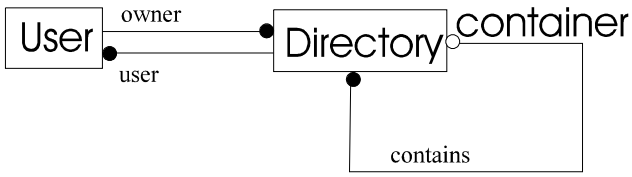
1 A benützt 0 oder n B

1 A benützt 0 oder 1 B

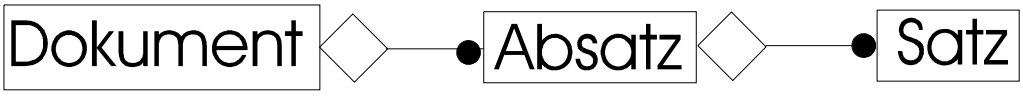
8.1.1.1.2 Assoziationsattribute



8.1.1.1.3 Rollen



8.1.1.2 Aggregation (hat-Beziehung)



8.1.1.3 Generalisierung (ist-Beziehung)

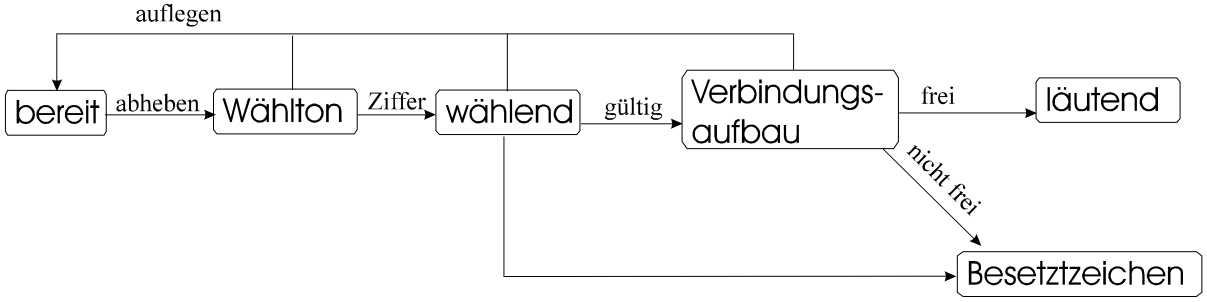
Vererbung

8.2 Dynamische Modell (Implementierung)

- Reihenfolge der Operationen
- Zustandsüberführungen
- **Ereignis:** Einwirkung auf ein Objekt, sodaß es seinen Zustand ändert oder andere Ereignisse auslöst
- **Szenario:** Folge von Ereignissen
- Ereignisdiagramm (Event Trace diagram)
- Zustand und Ereignis sind dual

8.2.1 Zustandsdiagramm

- Für jede Klasse eigenes Diagramm
- aus Zustand lassen sich verschiedene Szenarios ableiten
- endlicher Automat
- D.M = Menge von Zusandsdiagrammen

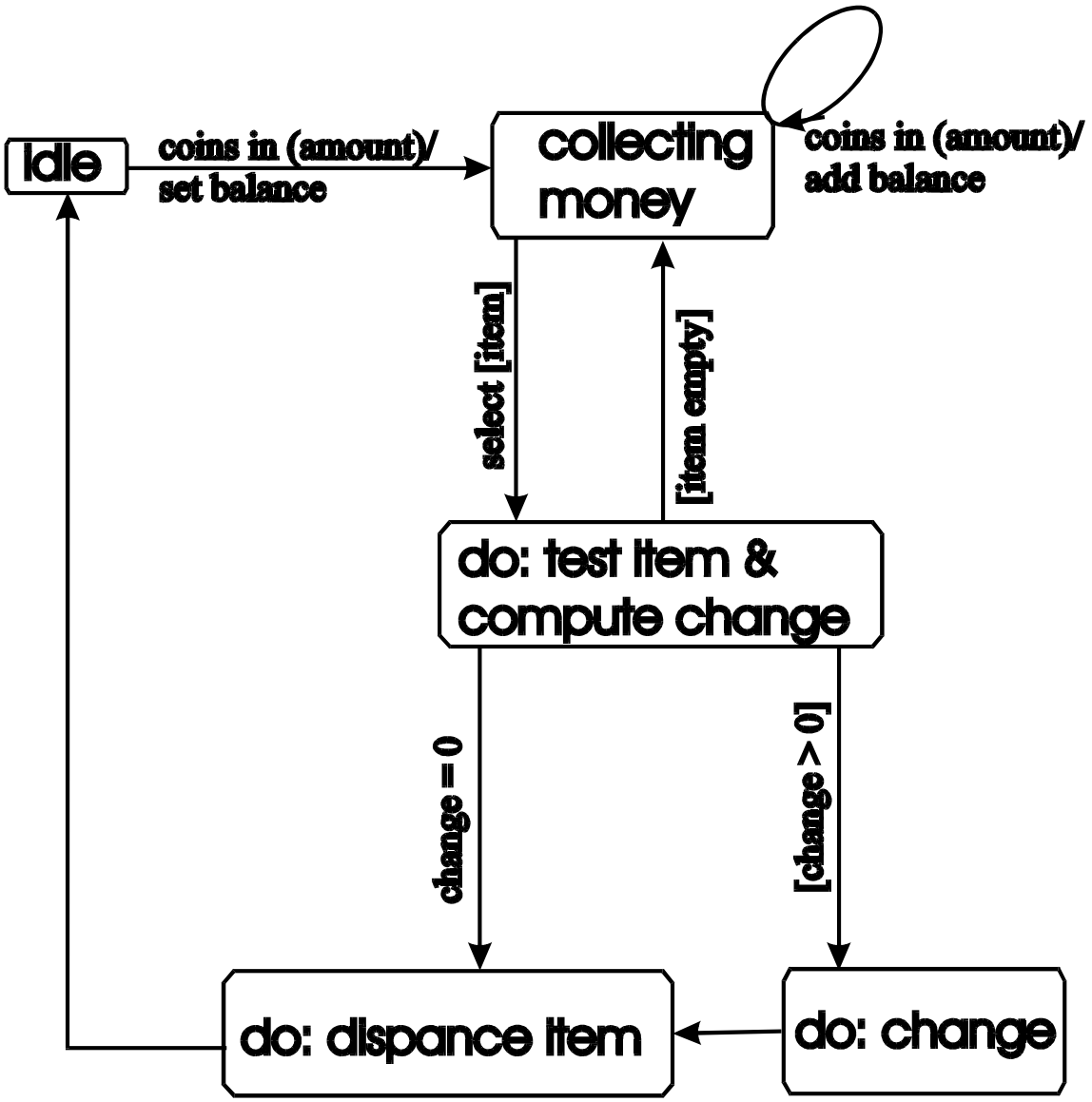


Zustandsübergänge können von Bedingungen abhängig sein : Ereignis [Bedingung]
Operationen begleiten Zustände und Ereignisse

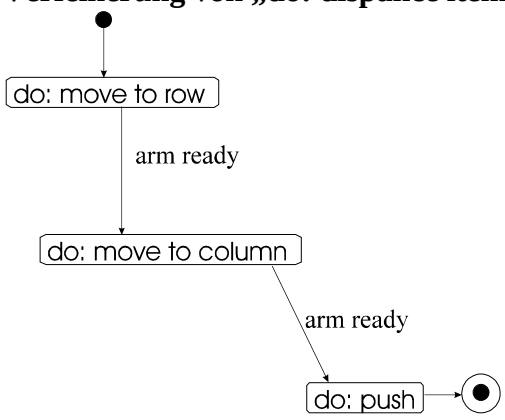
Aktivitäten	**Aktionen**
hat bestimmte Dauer	ohne nennenswerte Dauer
treten in Zuständen auf	werden durch Ereignisse ausgelöst
Zustand do: Aktivität	*Ereignis (Attribut) [Bedingung]/Aktion*

8.2.2 Geschachteltes Zustandsdiagramm

Verkaufsautomat:



Verfeinerung von „do: dispance item“:

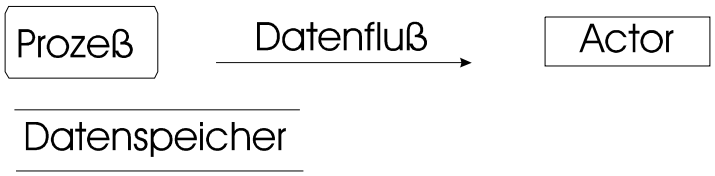


8.2.3 Parallelität

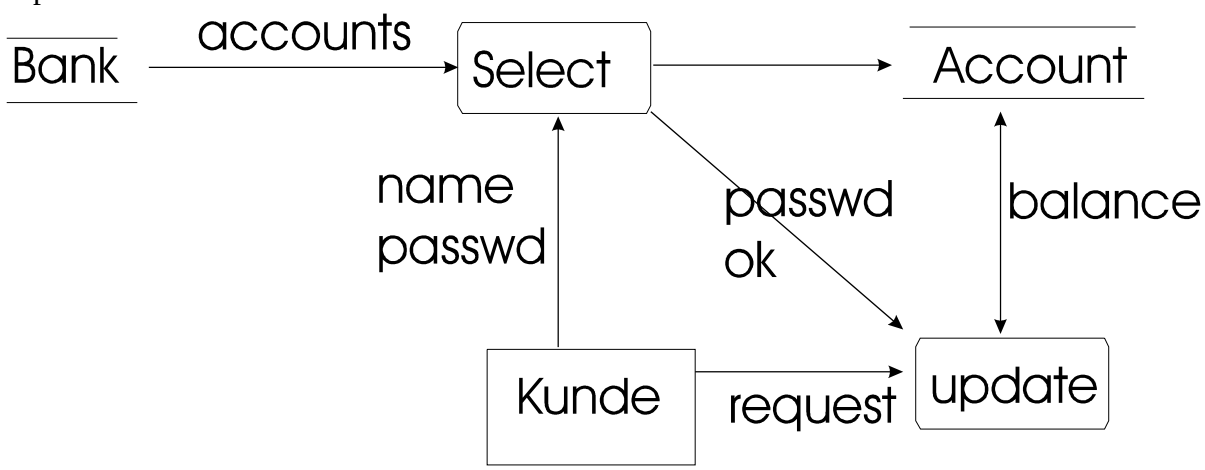
mehrere Zustandsdiagramme
Gesamtzustand = $Z_1 \times Z_2 \times ... Z_n$

8.3 Funktionales Modell (Datenfluß)

Menge von Datenflußdiagrammen



Bsp.:



9 Klassenbibliotheken

9.1 Kategorien von Klassen

9.1.1 Smalltalk-Klassenbibliothek

9.1.2 ET++-Klassenbibliothek

9.2 Werkzeuge für Klassenbibliotheken

9.2.1 Objektbrowser

9.2.2 ET++-Systemklassen

9.2.3 ET++-Datenstrukturenklassen

9.2.4 Graphische Klassen (Benutzerschnittstellenklassen)

10 Frameworks

10.1 Implementierung von Frameworks

10.2 Prozedurbibliotheken versus Frameworks

10.2.1 Framework für Menüs

10.2.2 Framework für Kommandos (Undo/Redo)

10.2.3 Framework für "Dragging" von Figuren

10.3 Model-View-Controller-Frameworks (MVC)

10.3.1 MVC in Oberon

10.3.2 Beispiel für Notifikation

10.3.3 MVC-Framework als Diagramm

10.3.4 Framework für Objekte in Texten

10.4 Application Frameworks

10.5 Herstellen einer leeren Applikation mit MacApp

11 Design Pattern

11.1 Was ist Design Pattern

Eine schematische Lösung für häufig wiederkehrendes Problem

Bsp.: Divide and Conquer
Wenn Problem klein genug, dann
löse es
sont wende das selbe Verfahren auf Teilprobleme an;
Kombiniere Resultate

Bestandteile

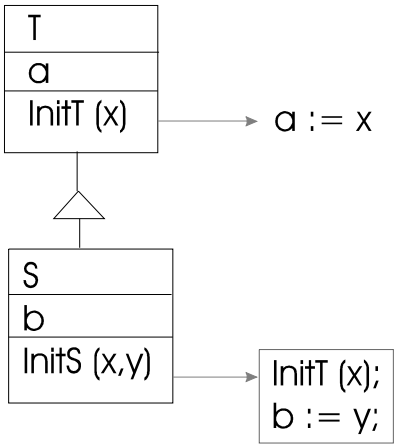
- Name: schafft Vokabular
- Problembeschreibung
- Lösung:
  - Komponenten
  - Operationen
  - Interaktionen

Frameworks	**Pattern**
wiederverwendbare Software	wiederverwendbares Wissen
groß	klein
domainspezifisch	allgemein

11.2 Erzeugende Muster

11.2.1 Constructor (New T)

11.2.1.1 Erzeugung und Initialisierung von Objekten



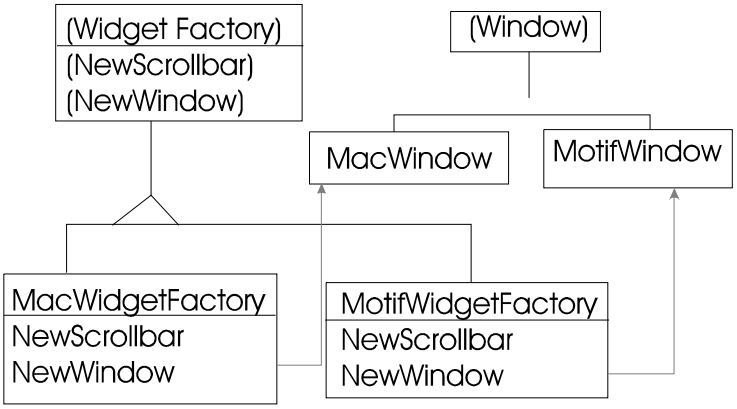
```
Procedure NewT (x ...) : T;
BEGIN
        NEW (t);
        t.InitT (x);
        RETURN t;
END NewT;

PROCEDURE NewS (x, y, ...) : S;
BEGIN
        NEW (s);
        s.InitS (x,y);
        RETURN s;
END NewS;

t := NewT (t);
s := NewS (x,y);
```

11.2.2 Erzeugung von Objekten variablen Typs

Editor → Window → MacWindow
Problem: Editor kann keine MacWindows erzeugen

11.2.2.1 Abstract Factory

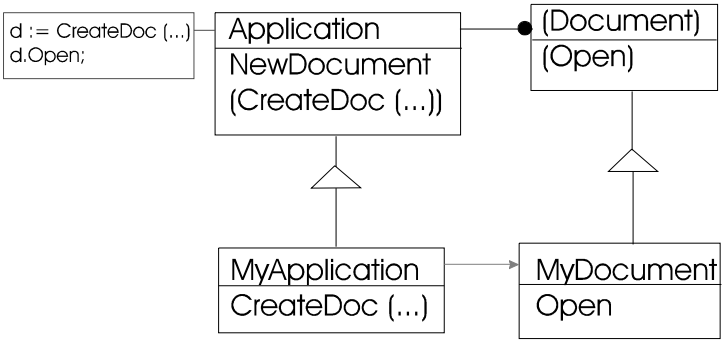
Erzeugung eines Objekts mit bestimmter Schnittstelle, wobei dynamischer Typ offen gelassen werden soll



Statt:
```
NEW (Window);
Window := WidgetFactory.NewWindow (...);
```
Initialisierung:
```
WidgetFactory := MacWidgetFactory;
```

11.2.2.2 Factory Method

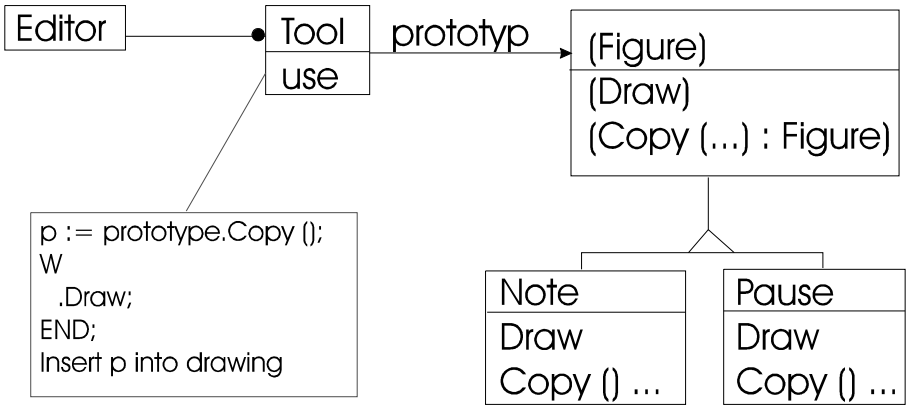
Erzeugung von Objekten soll Unterklassen überlassen werden



11.2.2.3 Prototype

Erzeugung eines Objekts als Kopie eines Musterobjekts

ad Bsp Noteneditor:



11.2.3 Bsp: Erzeugung eines Labyrinths

11.2.3.1 Factory Method

11.2.3.2 Prototype

11.2.3.3 Abstract Factory

11.2.3.4 Builder

Vorteilhaft:

- wenn Aussehen der zu erzeugenden Objekte erst zur Laufzeit bekannt
  - vom Benutzer zusammengesetzt
  - dynamisch geladen
- wenn lauter gleich initialisierte Objekte erzeugt werden sollen
- wenn es sonst zu viele Factories gäbe

11.3 Strukturmuster

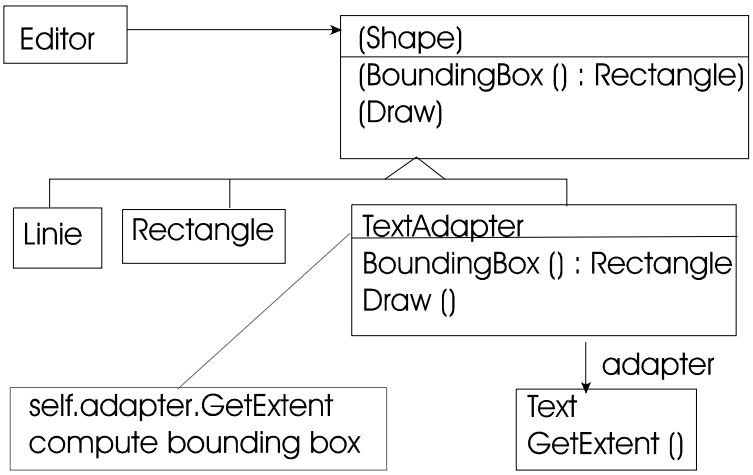
Muster aus mehreren Klassen

11.3.1 Familie

Abstrakte Klasse + Unterklasse

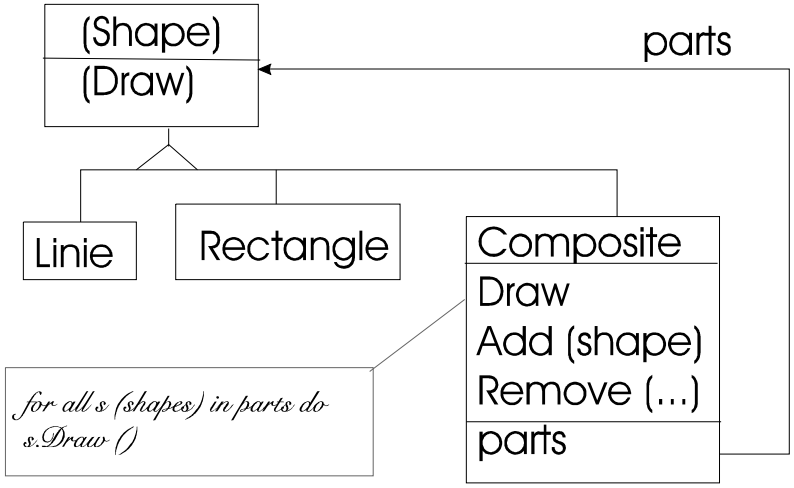
11.3.2 Adapter (oder Wrapper)

Eine Klasse kompatibel zu einer anderen machen.



11.3.3 Composite

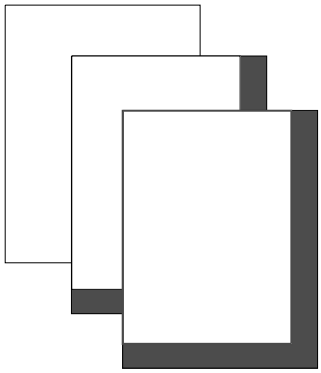
Herstellung zusammengesetzter Objekte, die wie Einzelobjekt behandelt werden können



11.3.4 Decorator

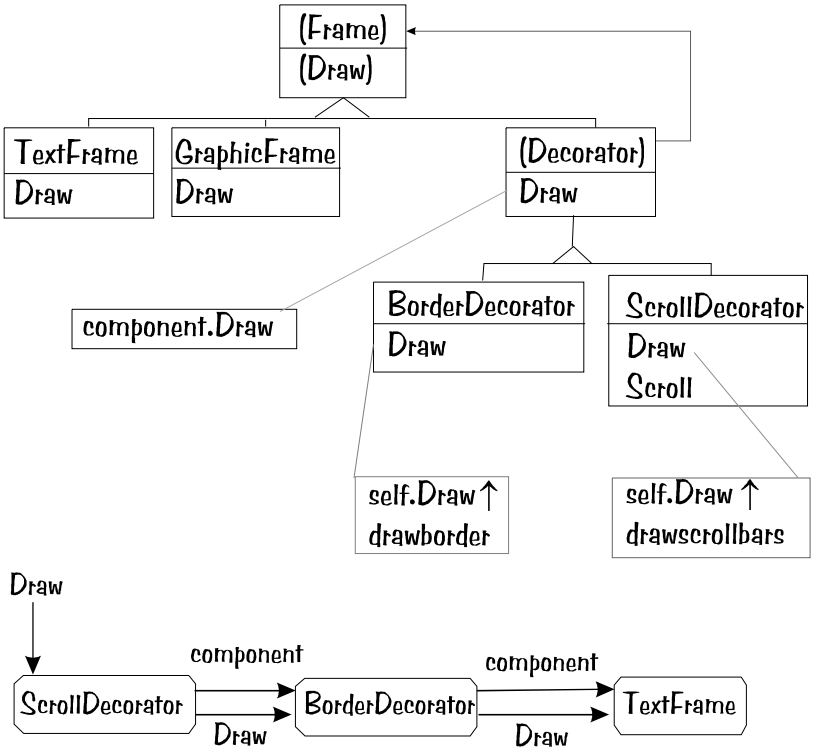
Neues Verhalten hinzufügen, ohne Unterklassen zu bilden

Bsp.: Hinzufügen von Scollbars und Rand zu einem Textframe



Möglichkeiten:

- Unterklassen: ScollBarTextFrame
- Objekt davorschalten, das Scrollbars hinzufügt



transparent: Klient merkt nicht, daß Decorator davorgeschaltet ist

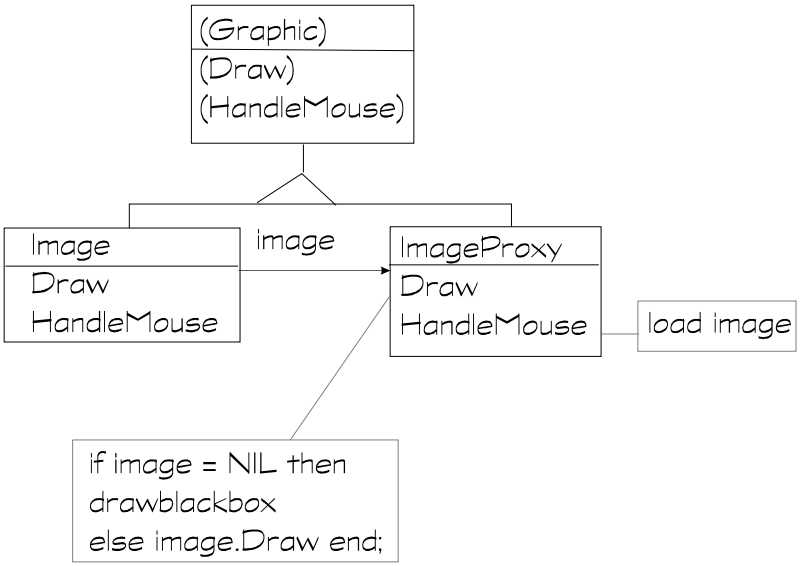
Flexibler als Vererbung, da
- keine Unterklassen nötig
- Funktionalität beliebig (tief) zusammensetzbar
- Funktionalität zur Laufzeit änderbar

Nachteile:
- langsamer
- kein direkter Zugriff auf Attribute mehr möglich, da Objektcharakter verändert wird

11.3.5 Proxy

Platzhalter für anderes Objekt

Bsp.: Text mit Bildern, die zunächst als BlackBox erscheinen sollen.



Client → (request) Proxy → (request) RealObject

Weiter Beispiele

- Remote Proxy
- Protection Proxy
- Trace Proxy

Nachteile (siehe Decorator)

11.3.6 Fassade

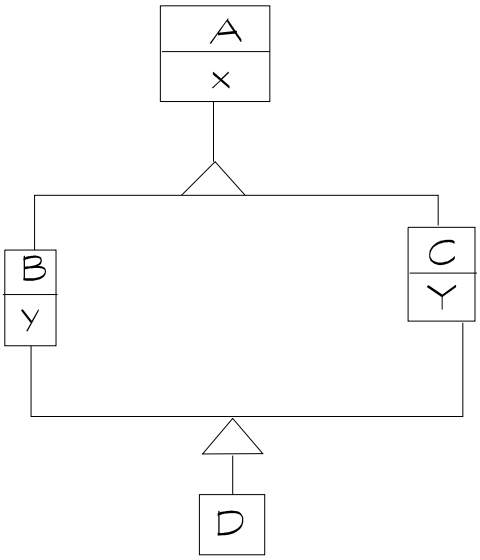
Ansprechpartner für Subsytem; liefert die Schnittstelle zu mehreren Unterklassen, sodaß sich der Client lediglich an eine Schnittstelle halten muß.

11.3.7 Twin

Zur Modellierung mehrfacher Vererbung

Problem:

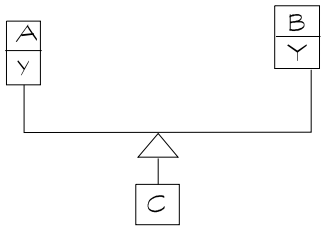
- Namenskonflikte: gleichnamig geerbte Felder
- Rautenstruktur



- Klassenbibliothek wird unübersichtlich
- Ineffizienz

11.3.8 Vermeidung mehrfacher Vererbung

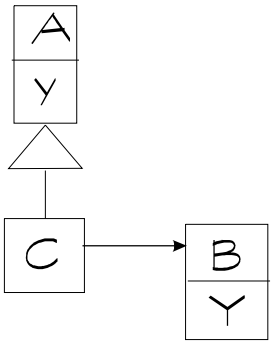
11.3.8.1 Umstrukturieren der Klassenhierarchie



$\Rightarrow A \rightarrow B \rightarrow C$ oder $B \rightarrow A \rightarrow C$

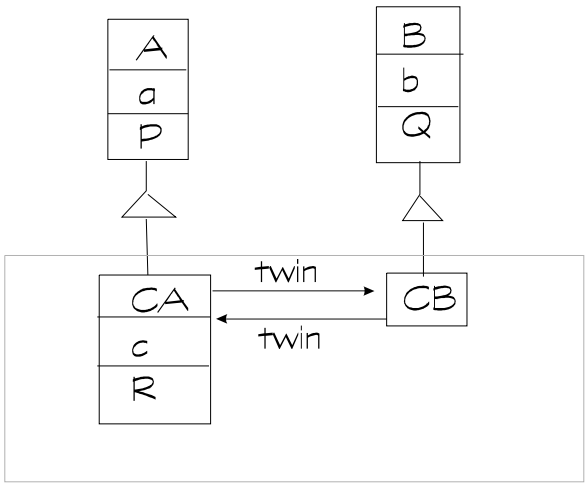
oft nicht möglich!

11.3.8.2 Adapter



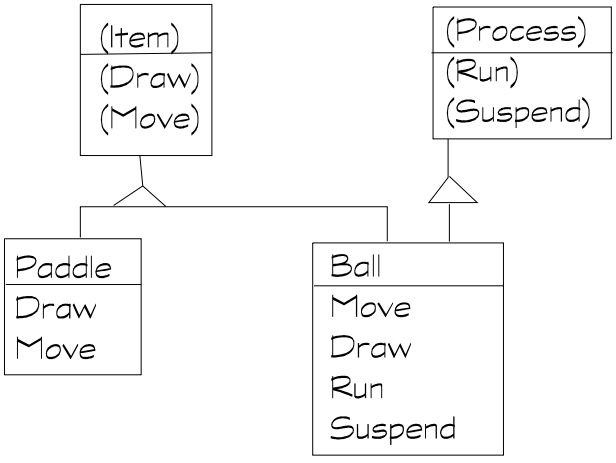
C.x;C.b.y;
Nachteil: aber C ist nicht mehr mit B kompatibel!

11.3.8.3 Twin

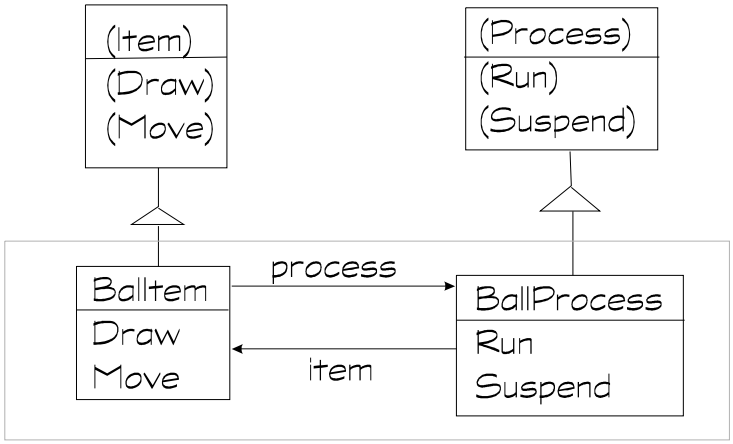


- über CA-Ende mit A-Objekt kompatibel
- über CB-Ende mit B-Objekt kompatibel
- keine Probleme mit Namenskonflikten

Bsp.: Ballspiel; Bälle sind aktive Spielgegenstände



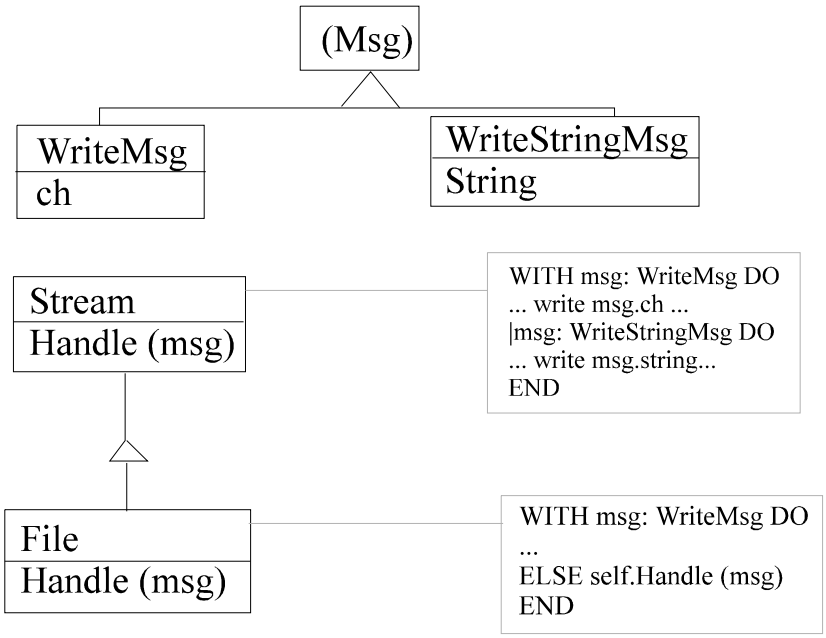
⇒



11.4 Verhaltensmuster

Modellieren, Interaktion zwischen Objekten

11.4.1 Meldungsobjekte



Reaktionen auf unbekannte Meldungen

- Fehler melden
- ignorieren
- an anderes Objekt weiterleiten (Delegation)

Handle als Prozedurvariable möglich

⇒ Reaktion auf Meldungen zur Laufzeit austauschbar

Vergleich: Meldungsobjekte / Methoden

+ flexibler: Meldungen speichern, übers Netz schicken
+ Reaktionen auf unbekannte Meldungen
+ austauschbares Verhalten
+ Meldungen können hinzukommen, ohne daß sich die Klassenschnittstelle ändert
- erlaubte Meldungen sind nicht aus Klassenschnittstellen ersichtlich
- keine statische Meldungsprüfung (Fehler tritt dynamisch vielleicht erst nach Monaten auf)
- langsam
- umständlich: *msg.ch = "x"; file.Hanlde (msg)*

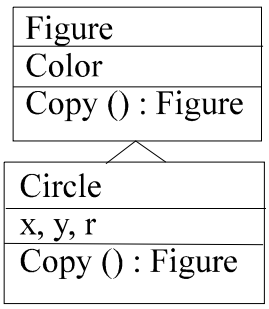
Zu empfehlen bei

- Broadcasts
- Schlupfloch für unbekannte (neue) Meldungen

11.4.2 Copy

Kopieren eines Objekts, dessen dynamischen Typ man nicht kennt

```
VAR f, g: Figure; (* dynamischer Typ: Rectangle *)
g := f;
g↑ := f↑;          (* da g noch nicht existiert *)

g := f.Copy ();
```



Problem: bei Copy von Circle: *c := NewCircle (x, y, r)* werden die Basisattribute nicht mit kopiert! ⇒

```
g := NIL; f.Copy (g);
```

Copy von Figure wird zu Copy (VAR g : Figure) : Figure

```
IF g = NIL THEN NEW (g); END;
g.color := self.color;
```

Copy von Circle wird zu Copy (VAR g : Figure) : Figure

```
IF g = NIL THEN
        NEW (c);
        g := c;
END;
WITH g : circle DO
        g.x := self.x;
        g.y := self.y;
        g.r := self.r;
END;
self.Copy↑ (g);
```

jedoch ist die vorhergehende (notwendige) Anweisung *g := NIL* sehr unschön ⇒

```
f.Copy (g);
```

Copy von Figure:

```
type := Types.TypeOf (self);
Types.NewObj (type, g);
g.color := self.color;
```

Copy von Circle:

```
self.Copy↑ (g);
WITH g : circle DO
        g.x := self.x;
        ....
END;
```

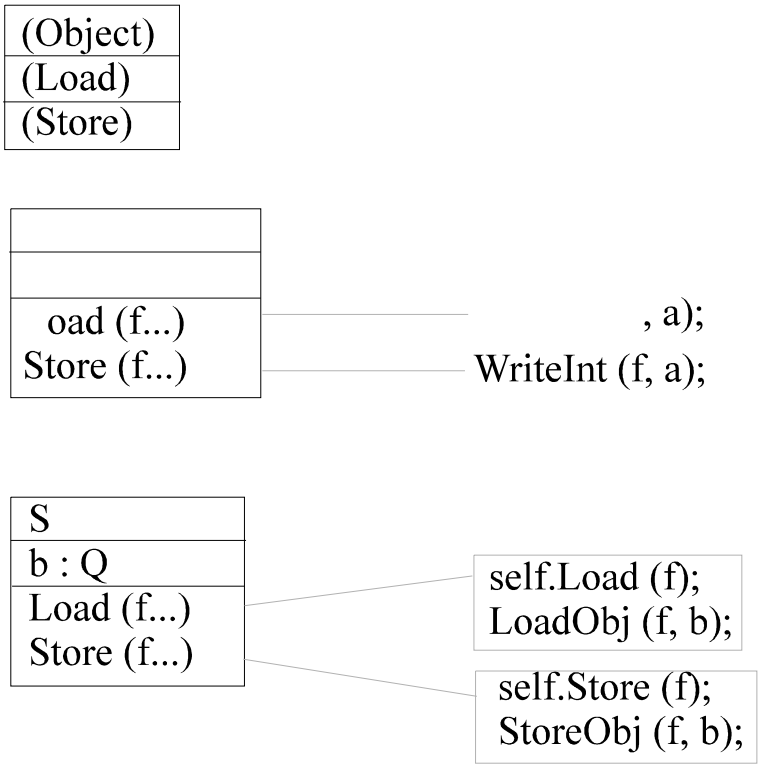
11.4.3 INPUT / OUTPUT

Ein/Ausgabe von Objekten, deren dynamischen Typ man nicht kennt

Problem: Beim Lesen muß Objekt erzeugt werden, aber von welchem Typ?

Lösung: Typnamen mit rausschreiben

```
PROCEDURE StoreObj (f : File; obj: Object);
BEGIN
        type = Types.TypeOf (obj);
        WriteString (f, type.name);
        obj.Store (f);
END StoreObj;

PROCEDURE LoadObj (f : File; VAR obj : Object);
BEGIN
        ReadString (f, name);
        type := Types.This (name);
        Types.NewObj (type, obj);
        obj.Load (f);
END LoadObj;
```



11.4.4 Iterator

Iterieren über Objektmenge, deren Aufbau man nicht kennt.
Bsp.: Drucken aller Elemente einer Liste

11.4.4.1 PrintAll

```
FOR all nodes x DO
        x.Print;
END;
```

11.4.4.2 Broadcast mit Meldungsobjekten

umständlich

11.4.4.3 Operation als Prozedurvariable

```
FOR all nodes x DO
        procVar (x);
END;

list.ForAll (Print);
list.ForAll (Store);

PROCEDURE Print (x : Node);
BEGIN
        x.Print;
END Print;
```

12 Smalltalk

PARC (Forschungslabor), Entwickler: Alan Key

12.1 Besonderheiten

- vollständig objektorientiert
- alle Datentypen sind Klassen (auch Integer, ...)
- alle Operationen sind Meldungen (auch +, >, ...)
- Ablaufkontrollstrukturen sind Meldungen
- Klassen sind Objekte
- konstante Ladestücke sind Objekte
- keine statischen Typen: nur dynamische Typenprüfung (zur Laufzeit)
- wird oft interpretiert: Bsp: Anhängen einer Meldung zur Laufzeit bei Oberon nicht möglich → Austausch des Meldungsinterpretors
- sehr umfangreiche Klassenbibliotheken

12.2 Klassen

12.3 Programme

Image: Menge aller Klassen
Meldungen an Klasse:

s := stack new : 100 (neues Stack Element)

Meldung an Objekt:

s push : 3 s push : 7 s push (s.pop + s.pop)

12.4 Ausdrücke

Konstanten - Objekte
Zahlen 3, -1.25
Zeichen $a, $+
Strings 'abc'
Arrays #(1 2 3), #(9 $9 nine #(0 1))
Variablen (enthalten Objekte, aber keine statischen Typ)
- Zeiger auf Objekt

12.4.1 Typen

Variablen haben dynamischen Typ
Deklaration:
|n|
n := 3 (dynamischer Typ = Integer)
n := 'abc' (dynamischer Typ = String)
keine statischen Typprüfungen
Typprüfung nur zur Laufzeit, wenn Meldung geschickt wird
'abc' + 'def' (zur Laufzeit: String versteht Meldung + nicht → ERROR)

12.4.2 Meldungen (3 Arten)

- unäre Meldungen (keine Parameter)

x factorial sqrt (x zuerst faktorisiert, Ergebnis dann sqrt)
stack pop

- linäre Meldungen (Operator: 1-2 Zeichen)

3 + 2 (s mal Integer Meldung + mit Parameter 2)
i > j

- Schlüsselwortmeldungen (Schlüsselwort + mehrere Parameter)

Bsp: list add : val
list at : 3 put : 7

12.4.3 Prioritäten (Abarbeitung von Meldungen)

1. Klammer (höchste)
2. unäre
3. linäre
4. Schlüsselwort (niedrigste)

Bsp:
string at : size - 1 pat : (string at : 1)
Objekt 1: (string at : 1)
Objekt 2: string vor size
Objekt 3: Länge des Strings
Objekt 4: Länge - 1

Nummer der Objekte geben die Erzeugung an
list add : 1; add : 2; add : 3

12.4.4 Blöcke

[i := file get. j := file get. k := i + j] (Block)
Ausführen von Blöcken: *[] value*
formale Parameter: *block := [:par] i := i + (par*par)*
aktuelle parameter: *block value : 10*

12.4.4.1 Ablaufkontrollstrukturen

12.4.4.1.1 if

(a > b) if True : [max := a] ifFalse : [max := b]
max := (a > b) if True : [a] ifFalse [b]

12.4.4.1.2 while

[i <=b] whileTrue : [list at : i put : 0 : i := i +1]

12.4.4.1.3 for

1 to : 10 do : [: i | sum := sum + i]

12.4.4.2 Iteratoren

12.4.4.2.1 do: "Zähler von Vokalen"

Kommentar unter ""
|n|
n := 0
string do : [: ch]
ch isVocal ifTrue : [n := n + 1]

12.4.4.2.2 select: "Zähler von Vokalen"

|n|
n := (string select : [: ch | ch isVocal]) size

12.4.4.2.3 collect: "Quadrate eines Arrays"

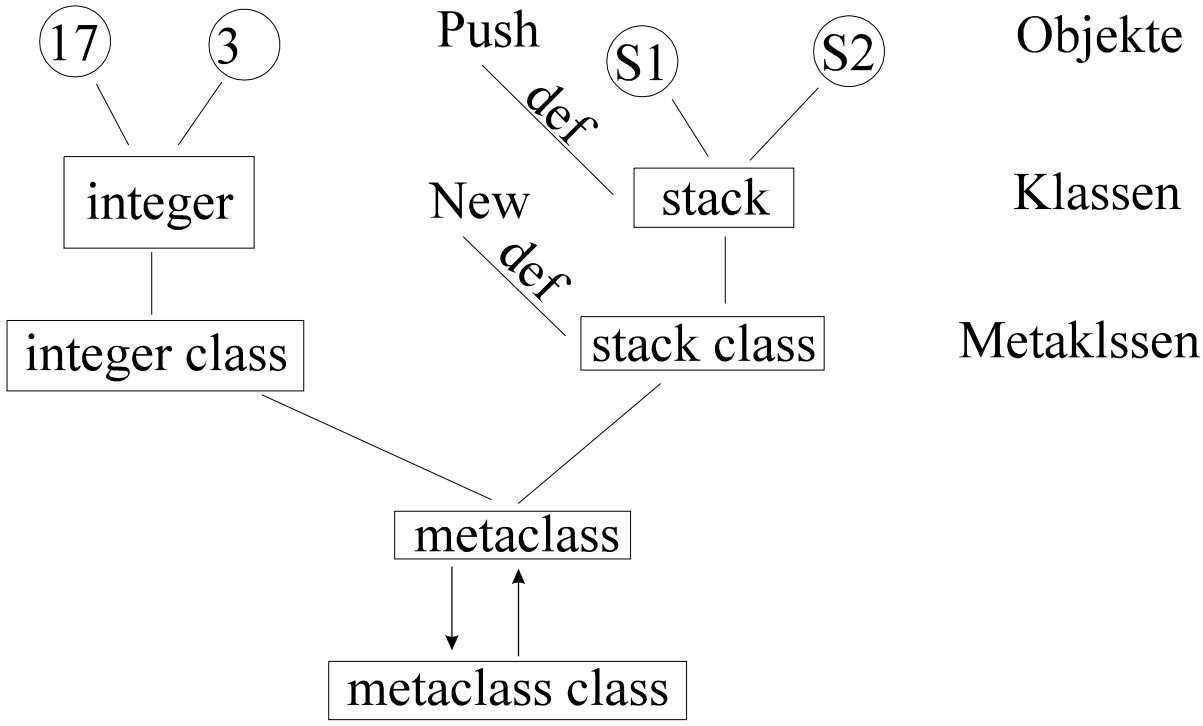
|squares|
*squares := #(1 2 3 4 5) collect [: i | i*i]*
Ergebnis: *squares = #(1 4 9 16 25)*

12.4.5 Metaklassen

Klasse einer Klasse

- Klassenvariablen: Attribute, globale Variablen
- Klassenmethoden: Operationen auf Klassen Bsp: *new* leifert Objekt

Bsp: Klasse von Stack heißt "Stack class" (Konvention)



13 C++

13.1 Klassen

13.2 Vererbung

13.3 Dynamische Bindung

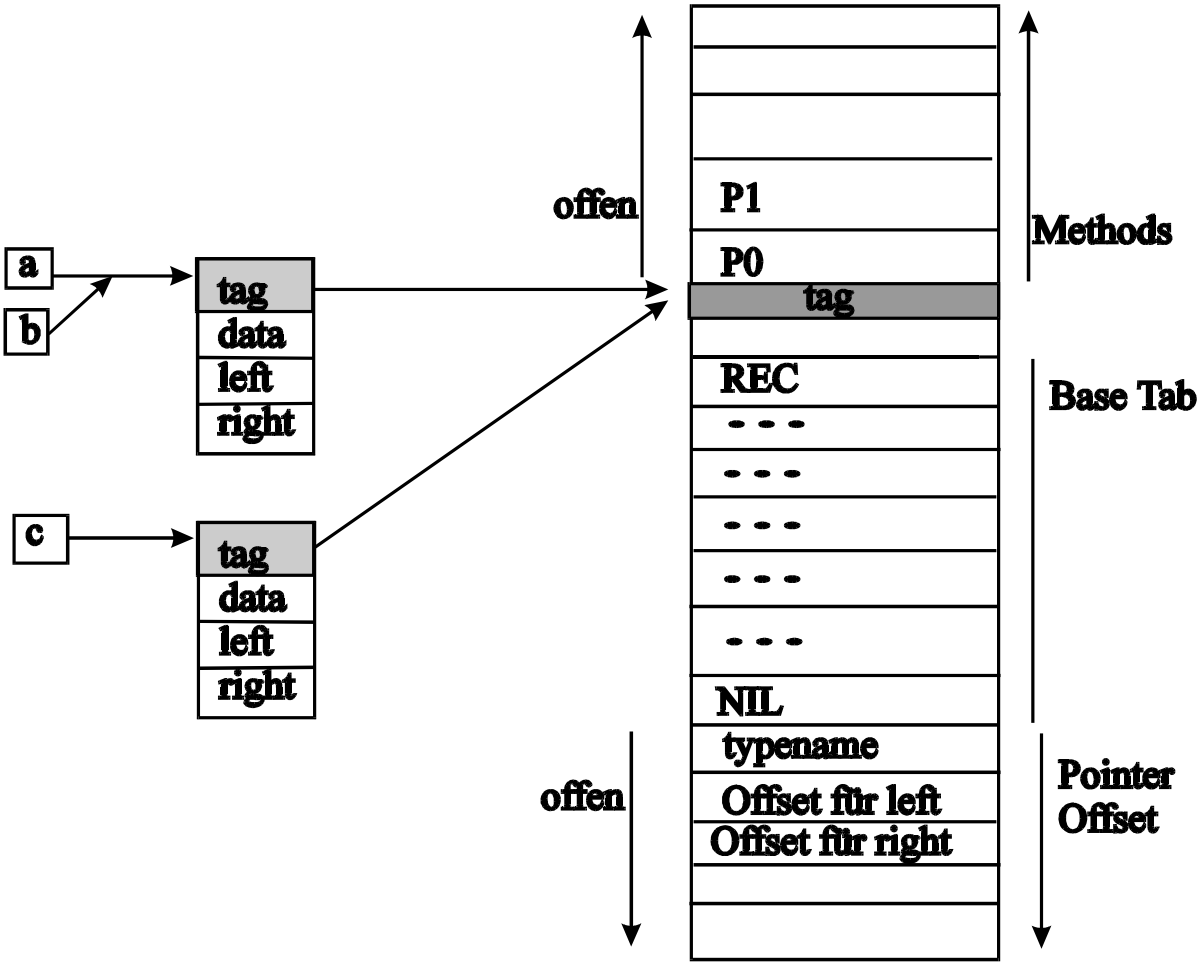
13.4 Templates

14 Implementierung objektorientierter Sprachen

- Datenstrukturen für Objekte und Klassen
- Meldungssuche

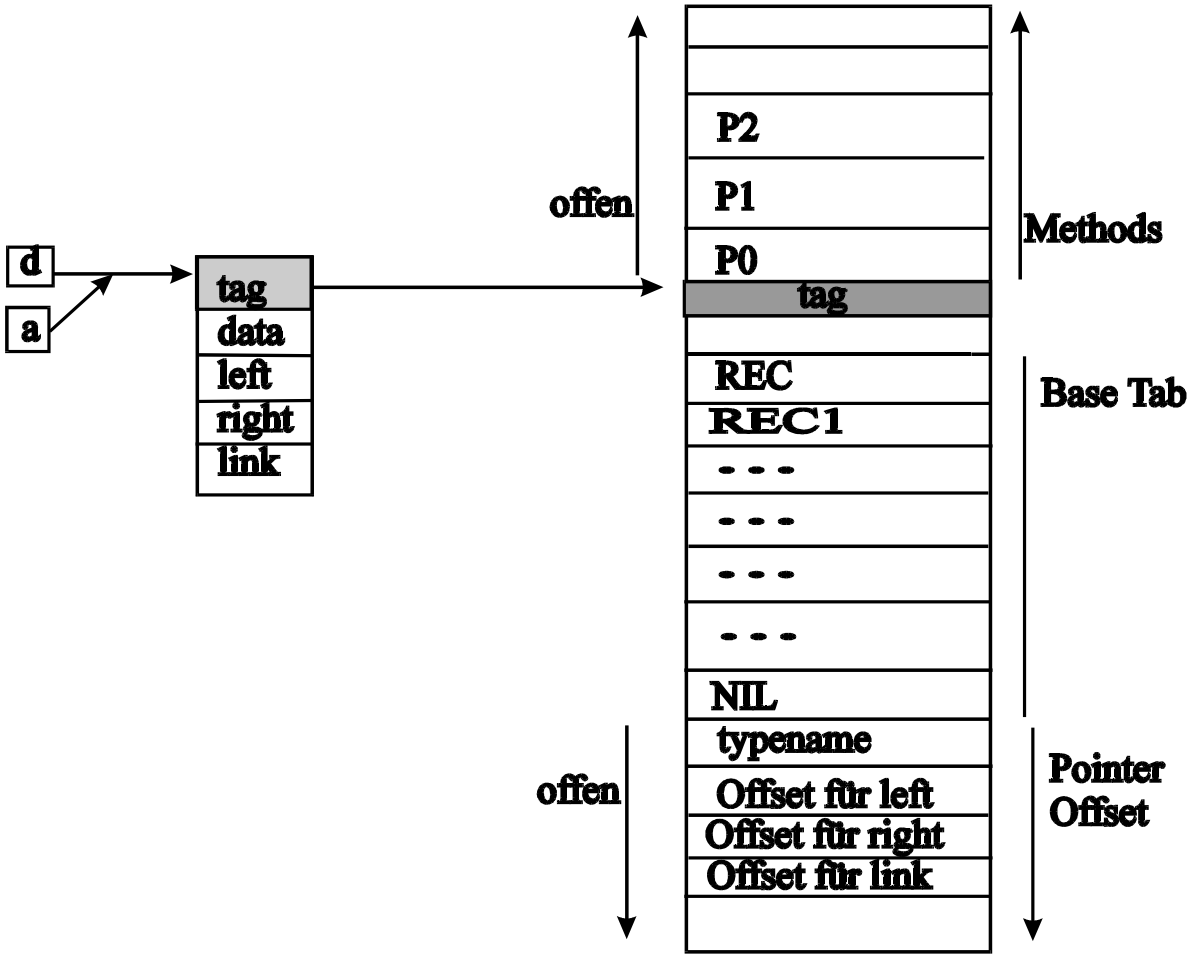
14.1 Oberon-2

```
TYPE Ptr = POINTER TO REC;
        REC = RECORD
                data: INTEGER;
                left, right : Ptr;
                PROC (p : Ptr) P0;
                PROC (p : Ptr) P1;
        END;

NEW (a);
b := a;
NEW (c);
```



Tag = Zeiger auf Typdeskriptor
Typdeskriptor = Datenstruktur, die den Type beschreibt; pro Klasse 1mal!!!
BaseTab auf 7 beschränkt!!

```
TYPE    Ptr1 = POINTER TO REC1;
        REC1 = RECORD (REC)
                  lint : Ptr1;
                  PROC (p : Ptr1) P0;
                  PROC (p : Ptr1) P2;
        END;

VAR d : Ptr1;
a := d
```



Methodenaufruf

a.P0; $\Rightarrow$ call a.tag.Methods[$Index_{P0}$]

a.tag.Methods[$Index_{P0}$] ist zur Compilezeit bekannt!

Typtest

IF a IS Ptr1 THEN ... $\Rightarrow$ IF a.tag.BaseTab[$Level_{Ptr1}$] = TD_{REC} THEN ...

a.tag.BaseTab[$Level_{Ptr1}$] ist zur Compilezeit bekannt!

15 Glossar objektorientierter Begriffe

Inhaltsverzeichnis zur Lehrveranstaltung

1 EINFÜHRUNG 2

1.1 HAUPTIDEEN VON OOP 2
1.1.1 Menge kooperierender Objekte 2
1.1.2 Arbeiten mit Varianten, ohne diese zu unterscheiden 2
1.1.2.1 Programmstruktur 2
1.1.2.1.1 Prozedural 2
1.1.2.1.2 Objektorientiert 3
1.1.2.2 Operationen 3
1.1.2.2.1 Prozedural 3
1.1.2.2.2 Objektorientiert 3
1.1.2.3 Merkmale objektorientierter Sprachen 3
1.1.2.3.1 Datenabstraktion 3
1.1.2.3.2 Vererbung (Typerweiterung) 4
1.1.2.3.3 Dynamische Bindung (Polymorphismus) 4
1.1.2.4 Terminologie 4
1.1.2.5 Geschichte von Object-Oriented-Programming-Languages 4

2 KLASSEN 5

2.1 IMPLEMENTIERUNG IN OBERON-2 5
2.1.1 Benutzung von Klassen 5
2.2 OBJEKTE UND REFERENZEN 5

3 VERERBUNG 6

3.1 NOTATION 6
3.2 ÜBERSCHREIBEN GEERBTER METHODEN (OVERRIDING) 6
3.3 BENUTZUNG 7
3.4 KLASSENHIERARCHIEN 7
3.5 TYPKOMPATIBILITÄT 7
3.6 ZUWEISUNG 8
3.6.1 statischer Typ 8
3.6.2 dynamischer Typ 8
3.7 RECORDZUWEISUNGEN 9
3.8 VARIABLENINHALT 9
3.9 TYPTEST ZUR LAUFZEIT 9
3.10 TYPZUSICHERUNG ZUR LAUFZEIT (TYPE GUARD) 10
3.11 WITH-ANWEISUNG 10
3.12 MEHRFACHE VERERBUNG 10

4 DYNAMISCHE BINDUNG 11

4.1 ABSTRAKTE KLASSEN 12

5 KONTRAKTE 12

5.1 KLASSENASSERTIONEN 13
5.2 SUBKONTRAKTE 13
5.3 ASSERTIONEN IN UNTERKLASSEN 14
5.4 PARAMETERTYPEN BEIM ÜBERSCHREIBEN 14

6 TYPISCHE ANWEDUNGEN VON OOP 15

6.1 DATENABSTRAKTION 15
6.1.1 Andere Abstraktionsmittel als Klassen 15
6.2 HETEROGENE DATENSTRUKTUREN 15
6.2.1 Konventionelle Implementierung 16
6.2.2 Objektorientierte Implementierung 16
6.2.3 Erweiterbarkeit zur Laufzeit 16
6.2.4 Weitere Beispiele für heterogene Datenstrukturen 17

6.3 GENERISCHE BAUSTEINE 17
6.3.1 Implementierung 17
6.3.2 Echte Generizität 18
6.4 AUSTAUSCHBARES VERHALTEN 18
6.5 ERWEITERUNG UND ANPASSUNG VON BAUSTEINEN 18

7 OBJEKTORIENTIERTER ENTWURF 19

7.1 PROZEDURORIENTIERTER ENTWURF 19
7.2 OBJEKTORIENTIERTER ENTWURF 19
7.3 ALLGEMEINE VORGEHENSWEISE 19
7.3.1 Beispiel einer objektorientierten Zerlegung 20
7.4 METHODE VON ABBOTT 20
7.4.1 Probleme der Methode von Abbott 21
7.4.2 Aufgabenstellung: Bankomatsoftware 21
7.5 CRC-KARTEN (CLASS-RESPONSIBILITIES-COLLABORATORS) 22
7.6 BEZIEHUGEN ZWISCHEN KLASSEN 22
7.6.1 Collaboration-Graph 22
7.7 HÄUFIGE ENWURFSFEHLER 22
7.7.1 Verwechslung von Ist-Beziehung und hat-Beziehung 23
7.7.2 Verwechslung von Oberklasse und Unterklasse 23
7.7.3 Falsches Empfängerobjekt 23
7.8 EINIGE TIPS 23
7.8.1 Wann sind Klassen sinnvoll 24

8 OBJECT MODELLING TECHNIQUE (OMT) 25

8.1 OBJEKTMODELL 25
8.1.1 Beziehungen 25
8.1.1.1 Assoziationen (benutzt-Beziehung) 25
8.1.1.1.1 Anzahl der Beziehungen 26
8.1.1.1.2 Assoziationsattribute 26
8.1.1.1.3 Rollen 26
8.1.1.2 Aggregation (hat-Beziehung) 26
8.1.1.3 Generalisierung (ist-Beziehung) 26
8.2 DYNAMISCHE MODELL (IMPLEMENTIERUNG) 27
8.2.1 Zustandsdiagramm 27
8.2.2 Geschachteltes Zustandsdiagramm 28
8.2.3 Parallelität 29
8.3 FUNKTIONALES MODELL (DATENFLUß) 29

9 KLASSENBIBLIOTHEKEN 30

9.1 KATEGORIEN VON KLASSEN 30
9.1.1 Smalltalk-Klassenbibliothek 30
9.1.2 ET++-Klassenbibliothek 30
9.2 WERKZEUGE FÜR KLASSENBIBLIOTHEKEN 31
9.2.1 Objektbrowser 31
9.2.2 ET++-Systemklassen 31
9.2.3 ET++-Datenstrukturenklassen 31
9.2.4 Graphische Klassen (Benutzerschnittstellenklassen) 31

10 FRAMEWORKS 32

10.1 IMPLEMENTIERUNG VON FRAMEWORKS 32
10.2 PROZEDURBIBLIOTHEKEN VERSUS FRAMEWORKS 32
10.2.1 Framework für Menüs 32
10.2.2 Framework für Kommandos (Undo/Redo) 33
10.2.3 Framework für "Dragging" von Figuren 33
10.3 MODEL-VIEW-CONTROLLER-FRAMEWORKS (MVC) 33
10.3.1 MVC in Oberon 34
10.3.2 Beispiel für Notifikation 34
10.3.3 MVC-Framework als Diagramm 34

10.3.4 Framework für Objekte in Texten *34*
10.4 APPLICATION FRAMEWORKS 35
10.5 HERSTELLEN EINER LEEREN APPLIKATION MIT MACAPP 35

11 DESIGN PATTERN 36

11.1 WAS IST DESIGN PATTERN 36
11.2 ERZEUGENDE MUSTER 36
11.2.1 Constructor (New T) *36*
11.2.1.1 Erzeugung und Initialisierung von Objekten 36
11.2.2 Erzeugung von Objekten variablen Typs *37*
11.2.2.1 Abstract Factory 37
11.2.2.2 Factory Method 38
11.2.2.3 Prototype 38
11.2.3 Bsp: Erzeugung eines Labyrinths *39*
11.2.3.1 Factory Method 39
11.2.3.2 Prototype 39
11.2.3.3 Abstract Factory 39
11.2.3.4 Builder 39
11.3 STRUKTURMUSTER 40
11.3.1 Familie *40*
11.3.2 Adapter (oder Wrapper) *40*
11.3.3 Composite *41*
11.3.4 Decorator *41*
11.3.5 Proxy *43*
11.3.6 Fassade *43*
11.3.7 Twin *44*
11.3.8 Vermeidung mehrfacher Vererbung *44*
11.3.8.1 Umstrukturieren der Klassenhierarchie 44
11.3.8.2 Adapter 45
11.3.8.3 Twin 45
11.4 VERHALTENSMUSTER 46
11.4.1 Meldungsobjekte *46*
11.4.2 Copy *47*
11.4.3 INPUT / OUTPUT *48*
11.4.4 Iterator *49*
11.4.4.1 PrintAll 49
11.4.4.2 Broadcast mit Meldungsobjekten 49
11.4.4.3 Operation als Prozedurvariable 49

12 SMALLTALK 50

12.1 BESONDERHEITEN 50
12.2 KLASSEN 50
12.3 PROGRAMME 50
12.4 AUSDRÜCKE 50
12.4.1 Typen *50*
12.4.2 Meldungen (3 Arten) *51*
12.4.3 Prioritäten (Abarbeitung von Meldungen) *51*
12.4.4 Blöcke *51*
12.4.4.1 Ablaufkontrollstrukturen 51
12.4.4.1.1 if 51
12.4.4.1.2 while 51
12.4.4.1.3 for 51
12.4.4.2 Iteratoren 52
12.4.4.2.1 do: "Zähler von Vokalen" 52
12.4.4.2.2 select: "Zähler von Vokalen" 52
12.4.4.2.3 collect: "Quadrate eines Arrays" 52
12.4.5 Metaklassen *52*

13 C++ 53

13.1 KLASSEN 53
13.2 VERERBUNG 54

13.3 DYNAMISCHE BINDUNG 54
13.4 TEMPLATES 54

14 IMPLEMENTIERUNG OBJEKTORIENTIERTER SPRACHEN 56

14.1 OBERON-2 56

15 GLOSSAR OBJEKTORIENTIERTER BEGRIFFE 58