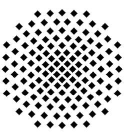
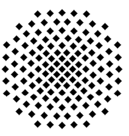


Objektorientierung in Ada 2005

Prof. Dr. Erhard Plödereder
Universität Stuttgart



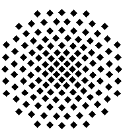
Polymorphie

Typpolymorphie: Die Fähigkeit einer benannten Einheit, Werte unterschiedlichen Typs zu besitzen.

Unterprogrammpolymorphie: Die Fähigkeit eines Unterprogramms, mehrere Implementierungen zu besitzen.

Typpolymorphie in Ada83

- “keine” (allenfalls Generic = Compilezeit-Polymorphie inkl. Ausprägen zu monomorphen Typen)
- für Variablen, Konstanten und Parameter: keine, nur monomorph
- für Parameter ererbter primitiven Operationen durch implizite Prä- und Postkonversionen bei der Übergabe simuliert

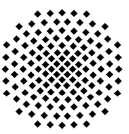


Typpolymorphie in Ada95

- jeweils beschränkt auf die Ableitungshierarchie (Klasse) eines "tagged type"

- für Variablen, Konstanten und Parameter:
  durch den "klassen-weiten Typ" T'Class

- für Parameter primitiver Operationen durch Referenz- und Viewsemantik der Übergabe



Typpolymorphie in Ada05

- jeweils beschränkt auf die Ableitungshierarchie (Klasse) eines "tagged type“ oder eines Interface

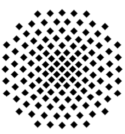
- für Variablen, Konstanten und Parameter:
  durch den "klassen-weiten Typ" T'Class

- für Parameter primitiver Operationen durch Referenz- und Viewsemantik der Übergabe

Primitive Operationen von Typen

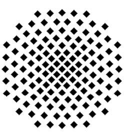
- ... sind Operationen, die implizit oder explizit in einer Paketspezifikation deklariert sind und einen im gleichen Paket deklarierten Typ in ihrer Signatur (Parameter oder Resultat) haben (“T” oder “**access** T”).
- Primitive Operationen werden von abgeleiteten Typen ererbt (durch implizite Deklaration mit der Typdeklaration) unter covarianter Ersetzung des alten durch den neuen Typ in ihrer Signatur.

  (entspricht den “Methoden” in anderen OOP-Sprachen.
  In der C++-Familie meist co-variantes Empfängerobjekt,
  aber invariante Parameter für ererbte Methoden.)



Ein Beispiel für Ada 83 Typableitung

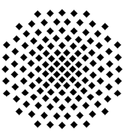
```
type Currency is ...;
function Interest(V: Currency; Future_Date: Date) return
   Currency;
type DM is new Currency;   -- a new type
  -- inherits 'function Interest(V:DM; Future_Date: Date) return DM;'
  -- the function could, but need not, be redefined here for type DM
type FF is new Currency;  -- ditto
DM_Account:  DM;
FF_Account: FF;
....Interest(FF_Account, New_Year);
     --- binds based on visibility, argument types, and overloading principles
FF_Account := DM_Account;   --illegal
```



Typenerweiterung (Ada95)

- "tagged" Typen können im Rahmen der Typableitung durch weitere sichtbare oder private Komponenten erweitert werden.

- primitive Funktionen, die den zugehörigen Typ als Resultatstyp haben, müssen im Rahmen einer solchen Typableitung redefiniert werden, d.h. es muß eine neue Implementierung vorhanden sein.
(Ausnahme: für abstrakte Typen, siehe später.)

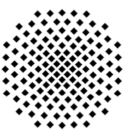


"Variations on a Theme"

```
type X is tagged private;

type Y is new X with private; -- can extend with private attributes
type Z is new Y with record  -- .... or visible ones
                              B: Some_Attribute;
                            end record;


type X is tagged record  A: Some_Attribute;  end record;

type Y is new X with private; -- can extend with private attributes
type Z is new Y with record  -- .... or visible ones
                              B: Some_Attribute;
                            end record;
```

Klassenweite Typen

- T`Class als **formaler Parametertyp**: akzeptiert aktuelle Parameter von beliebigem Typ, der von T abgeleitet ist.
- T`Class als **Zieltyp eines Zeigertyps**: Zeiger kann auf Objekte beliebigen Typs, der von T abgeleitet ist, zeigen.
- T`Class als **Typindikation in einer Objektdeklaration**:
  X: T`Class *-- ?? Ja, aber ...*
  1. Initialisierung erforderlich
  2. Initialisierung fixiert den "tag" für die Lebensdauer des Objekts auf den Typ der Erzeugung.
- **Generell: der "tag" eines Objekts ist unveränderlich.**
- **aber:** die statische Betrachtungsweise ("view") für ein Objekts kann auf Vatertypen seines tatsächlichen Typs beschränkt werden.



A User-Defined Class Hierarchy

ANIMAL

ANIMAL'CLASS

MAMMAL

MAMMAL'CLASS

PRIMATE

PRIMATE'CLASS

Klassenweite Typen

Beispiele für Kompatibilität:

```
A : ANIMAL;    -- monomorph
M : MAMMAL;
AC : ANIMAL'CLASS; -- polymorph
MC : MAMMAL'CLASS;
```

illegal

polymorphe Dekl. ohne Initialisierung

```
A := M;
M := A;
MC := AC; -- nicht Unterklasse
----------------------------
A := AC; -- … ausser bei der
A := MC;    Parameterübergabe
            an ein formales A
            in einer dispatch-
            ing operation
```

legal

```
AC := A;
AC := M;
AC := MC;
MC := M;
```

mit RT-checks:

```
A := ANIMAL'(AC);
MC := MAMMAL'CLASS(AC);
```

Klassenweite Access Typen

Beispiele für (fehlende) Kompatibilität:

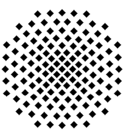
```
type Animal_Ptr is access Animal;
type Mammal_Ptr is access Mammal;
type Animal_Class_Ptr is access Animal'Class;
type Mammal_Class_Ptr is access Mammal'Class;
A : Animal_Ptr;   -- monomorph
M : Mammal_Ptr;
AC : Animal_Class_Ptr; -- polymorphes Ziel, aber
MC : Mammal_Class_Ptr; -- monomorpher Zeigertyp
```

illegal

Jegliche Zuweisungen oder Übergaben zwischen diesen Variablen oder Parametern

legal

Konventierungen, wenn Ziele konventierbar sind; Zuweisungen etc. nur mit anonymen access Typen



Klassenweite Access Typen

Konsequenz:

Der Versuch, eine polymorphe Zeigersemantik über die benannten, monomorphen Ada Access Typen zu realisieren, führt bei vielen Zuweisung oder Parameterübergaben zur Notwendigkeit expliziter Typkonversionen ! (oder zu hässlichen Konstrukten wie „X.all‘Access“)

Unterprogrammpolymorphie in Ada95

durch Ererbung und Redefinition primitiver Unterprogramme von "tagged types" in einer Ableitungshierarchie

Auswahl durch Dispatchingsemantik für kontrollierende Argumente von klassen-weitem (oder abstraktem) Typ

"Dispatching"

- Aufrufe sind **"dispatching"** wenn
  1. die aufgerufene Operation eine primitive Operation eines "tagged" Typs ist, und
  2. der "tag" des aktuellen Parameters nicht statisch bestimmbar ist (d.h., **der Parameter von klassen-weitem Typ** ist) oder **der aktuelle Parameter von abstraktem Typ** ist.
- **Dispatching** bedeutet, daß jene Implementierung der aufgerufenen Operation ausgeführt wird, die dem durch den "tag" des aktuellen Parameters identifizierten Typ zugeordnet ist.

"Dispatching" - Weitere Regeln

- Ein Unterprogramm darf nicht primitive Operation mehrerer "tagged" Typen sein.

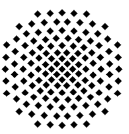
- Hat eine primitive Operation mehrere formale Parameter des zugehörigen Typs, führt ein Dispatching mit aktuellen Parametern, die unterschiedliche "tags" besitzen, zur Ausnahme.

(diese Regeln garantieren, daß beim Dispatching eine entsprechende, unter Covarianz ererbte Operation existiert.)

Interaktion mit Ada 83 Konzepten

- **Parameterübergabe:**
  Parameter von "tagged" Typ müssen per Referenz übergeben werden.
- **Typkonversion:**
  Erweiterung dieses Konzepts zur "Viewkonversion":
  Viewkonversion ändert die statische "Sicht" auf ein benanntes Objekt, produziert aber keine Kopie.
  D.h. Viewkonversion zum Vatertyp (insb. im Rahmen der Parameterübergabe) schneidet keine Komponenten ab, sie verhindert nur den Zugriff auf diese.

  (Ein Abschneiden erfolgt erst, wenn z.B. eine Zuweisung einen derartigen View als rechte Seite hat.)



Dispatching - Beispiel

type T **is tagged ...**
procedure Op1(X: **in out** T);

type NT **is new** T **with ....** -- *ererbt Op1*
procedure Op1(X: **in out** NT); -- *redefinierend*
procedure Op1(X: **in out** NT; Y: NT); -- *überladend*

X: T;
Y: T`Class := NT'(…) ;
Op1(X); -- *monomorpher aktueller Parameter => direkter Aufruf*
Op1(Y); -- *dispatching auf Op1 von NT*

Dispatching – erste Überraschung

package A **is**
  type T **is tagged ...**
  procedure Op1(X: **in out** T);
end A**;**

X: A.T;
Y: B.NT;
A.Op1(X); -- *direkter Aufruf*
A.Op1(Y); **-- illegal !**

package B **is**
  type NT **is new A.**T **with ....**
  procedure Op1(X: **in out N**T);
end B;

X: A.T’**class** := NT’(…);
Y: B.NT’**class :=** NT’(…);
A.Op1(X); -- *dispatching*
A.Op1(Y); **-- illegal ! (grundlos?)**

… eine Frage der Mehrdeutigkeit …

```
package A is
    type T is tagged null record;
    procedure Op1 (X : in out T);
    procedure Op1 (X : in out T'class);
end A;

  use A;
   X : T;
begin
   Op1(X);  -- illegal, da mehrdeutig !
```

Dispatching – erste Überraschung

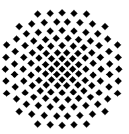
```
package A is
   type T is tagged ...
   procedure Op1(X: in out T);
end A;
```

```
package B is
   type NT is new A.T with ....
   procedure Op1(X: in out NT);
end B;
```

```
use A,B;
Y: NT;
Op1(Y); -- wäre mehrdeutig …
```

```
use A, B;
Y: NT'class :=  NT'(…);
Op1(Y);  -- wäre mehrdeutig …
```

… wenn A.Op1 zulässiger Kandidat wäre



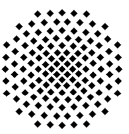
Dispatching – erste Überraschung

Konsequenzen:

- Man muss in Ada 95 die Methoden gemäß der statischen Typisierung genau ansprechen. Das ergibt viele unnötig erachtete „with“-Klauseln.

- Alternativ könnte man alle Pakete, die tagged Typen deklarieren, „use“n. Diese unnötige Öffnung von Sichtbarkeit ist aber von den meisten Styleguides verboten.

(Eine gute Lösung bietet erst Ada 2005 mit der Objektnotation.)



Redispatch ?

type T **is tagged ...**

procedure Op1(X: **in out** T);

procedure Op2(X: **in out** T) **is**
  begin ... Op1(X); -- *(re)dispatching ?* **NEIN !**

type NT **is new** T **with ....** -- *ererbt Op1, Op2*

procedure Op1(X: **in out** NT);

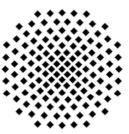
-- *procedure* *Op2(X:* ***in out*** *NT) ererbt*

Y: NT`Class := NT'(…) ;

Op2(Y); -- *dispatching*

Das allgemeine Redispatch-Problem

```
class C1
        method A ...
        method B ...
class C2 is new C1
        -- method A inherited
        method B is
                ... self.A ...  -- Annahme: dispatching semantics
O : C2;
...
O.B; -- ist das immer in Ordnung ?
```



Des Rätsels Lösung

```
class C1
        method A ... if Today=Thu then self.B;
        method B ...
class C2 is new C1
        -- method A inherited
        method B is
        ... self.A ...  -- dispatching semantics
O : C2;
...
O.B;  -- an Donnerstagen unendliche Rekursion !
```

... und die Konsequenzen ?

Welche Richtlinien kann man für die Programmierung erlassen, damit dieses Problem nicht auftritt ?

eine kritische, völlig offene Frage !!

aber: Redispatching wichtig für sogenannte "wrapper"-Methoden

Anmerkung: die Frage des rekursiven Aufrufs durch Redispatching ist selbst durch globale Programmanalysen nicht entscheidbar (nicht zuletzt, weil sie über verzeigerte Datenstrukturen in den Datenraum verschleppt wird und dort von den Graphstrukturen abhängt).

Redispatch in Ada95

kein Redispatch, außer für ...

- **Aufruf abstrakter primitiver Operationen:**

```
type NT is new T with ....  -- ererbt Op1, Op2
procedure Op1(X: in out NT) is   -- Redefinition
        begin   ....  Op2(X);   -- redispatch nur für abstraktes Op2
```

- **Bewusst programmierter Redispatch durch View-Konversion:**

```
procedure Op2(X: in out T) is
        begin ... Op1(T`Class(X)); --(re)dispatching
```

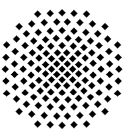
d.h., der für die Implementierung von T und dessen Op2 Verantwortliche entscheidet, nicht ein Automatismus.

- **Klassenweite Operationen redispatchen über dem Parameter:**

```
procedure P(X: in out T'Class) is  -- ist aber keine primitive Operation!
        begin ... Op2(T); -- dispatching
```

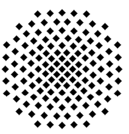
Abstrakte Typen und Unterprogramme

- Ein "tagged" Typ kann als "**abstrakt**" deklariert werden.
  (Anm.: "abstract" /= "private")
- Primitive Operationen abstrakter Typen können als abstrakt deklariert werden. Solche Operationen haben keinen Rumpf.
- **Regeln:**
  1. Es können keine Objekte abstrakten Typs erzeugt werden.
  2. Nicht-abstrakte Ableitungen abstrakter Typen müssen alle abstrakten ererbten Unterprogramme redefinieren.
  3. Aufrufe abstrakter Operationen müssen dispatching sein.
  4. primitive Funktionen, die den zugehörigen Typ als Resultatstyp haben, werden durch Ableitung abstrakt.



Case Analysis -- Expression Trees

```
type Expression_Node is abstract tagged record
     Source_Line : Positive;
     Source_Column: Positive;
end record;  -- objects cannot be of abstract type
type Node_Ptr is access Expression_Node'Class;

function Evaluate(Node: Expression_Node) return
   Float is abstract;  -- to be provided by derived types
procedure  Position_Cursor(Node: Expression_Node);
   -- inheritable implementation provided in package body, based on
   -- known attributes
```

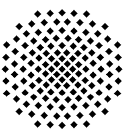


Expression Trees (con't)

```
type Literal_Node is new Expression_Node
   with record
         Value: Float;
   end record;

function Evaluate(Node: Literal_Node) return Float;
   --- returns Node.Value

type Binary_Exp is abstract new Expression_Node
    with record
         Left, Right: Node_Ptr;
    end record;
```

Expression Trees (con't)

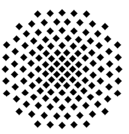
```
type Addition is new Binary_Exp with null record;
function Evaluate(Node: Addition) return Float;
    --- returns Evaluate(Node.Left.all) + Evaluate(Node.Right.all)

type Subtraction is new Binary_Exp with null record;
.....
etc.etc.

Root: Node_Ptr;
....
.... Evaluate(Root.all);  -- evaluates an expression tree of
                          -- arbitrary shape
```



Multiple Implementations: Integer Sets

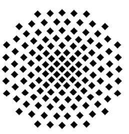
subtype Set_Element **is** Natural;
type Set **is abstract tagged null record;**

function Empty **return** Set **is abstract;**
function Unit(Element: Set_Element) **return** Set **is abstract;**
function Union(Left, Right: Set) **return** Set **is abstract;**
function Take(From: **in out** Set;
Element: **out** Set_Element) **is abstract;**
....-- *and so on*



Integer Sets: Bitvector Implementation

```
type Bit_Set is new Set
  with record
     Data: Bit_Vector;  -- declared elsewhere as array of boolean
   end record;

function Empty return Bit_Set;
    -- returns (Data => (others => False))
function Unit(Element: Set_Element) return Bit_Set;
    -- returns (Data => (Element => True, others => false)) appropriately constructed
function Union(Left,Right: Bit_Set) return Bit_Set;
    -- returns (Data => Left.Data or Right.Data)
function Take(From: in out Bit_Set; Element: out Set_Element);
   -- returns some random element
.....
-- easy to think of other implementations via type extensions.....
```

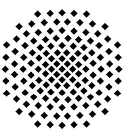


Integer Sets: Class-Wide Conversion

```
procedure Convert(From: Set'Class; To: out Set'Class) is
   -- convert between sets of different types and implementation
   Temp: Set'Class := From;
   Elem: Set_Element;
begin
   To := Empty;
   while Temp /= Empty loop
      Take(Temp, Elem);
      To := Union(To, Unit(Elem));
   end loop;
end Convert;
```

Reconfiguration/Extension: Devices

```
type Device is abstract tagged private;
    -- a device control block

procedure Open(Dev: in out Device) is abstract;
procedure Close(Dev: in out Device) is abstract;
...
-- these interfaces must be provided by specific device types

function Name(Dev: Device) return String;
...
-- non-abstract interfaces are inherited by derivative types,
-- and can be overridden
```

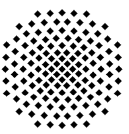


'Device' Example (con´t)

type Disk_Device **is new** Device **with ....** **--** *with additional attributes*

procedure Open(Dev: **in out** Disk_Device);
procedure Close(Dev: **in out** Disk_Device);
...
-- *inherited abstract interfaces must be provided by specific device types*

procedure Seek(Dev: Disk_Device, Track: Track_Number);
...
-- *additional interfaces applicable to Disk_Devices*

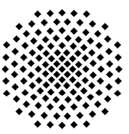


'Device' Example (con´t)

a device administrator might write:

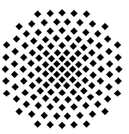
procedure Reset(Dev: **in out** Device´Class) **is**
-- this procedure is callable for objects of any type derived from Device
begin
Close(Dev); *-- these calls will "dispatch" to the appropriate implementation*
Open(Dev); *-- of Close and Open, depending on the specific type of Dev.*
end;

e.g.,
Disk: Disk_Device;
...
Reset(Disk); *-- will automatically call Open and Close for Disk_Device*



'Device' Example (con´t)

```
type Device_Ptr is access Device´Class;
type List_entry;
type Device_List is access List_Entry;
type List_entry is record
                        Dev: Device_Ptr;
                        Next: Device_List;
                   end record;
procedure Close_All(List:  in out Device_List) is
    Temp: Device_List := List;
begin
    while Temp /= null loop
      Close(Temp.Dev.all);    -- ... the specific device type is unknown.
      Temp := Temp.Next;      -- Close dispatches to the appropriate implementation.
    end loop;
end Close_all;
```

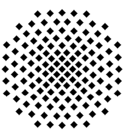


'Device' Example (con´t)

-- type Device_Ptr **is access** Device´Class;

type Device_ID **is private;**
-- with suitable operations to record a Device_ID in a Device
-- (control block) and to retrieve it from there.

procedure Install(Ptr: Device_Ptr; ID: Device_ID);
-- inserts pointer to device control block in OS table indexed
-- by device ID.
-- Subsequent calls to open (close, reset, etc.) a device given
-- the device ID will be turned into a call on the respective
-- operation of Ptr.all .
-- The abstract operations of type Device are effectively "call-outs"
-- from the OS.



Class-Wide Access Types

type T **is tagged record … ;**
type Ptr1 **is access** T;
type Ptr2 **is access** T'**class**;
procedure Check(X: T); -- primitiv für T, nicht für Ptr*!
Procedure Check(X: Ptr2); -- primitiv für monom. Ptr2

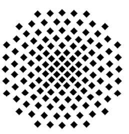
P1: Ptr1;
P2: Ptr2;

Check(P1); –- illegal (Typfehler)
Check(P2); – monomorph, daher nicht dispatching
Check(P1.all); – ok. Direkter Aufruf
Check(P2.all); – ok. Dispatching

Anonyme Access Typen

```
package A is
 type T is tagged record …;
 procedure check(X: T); -- primitiv für T
 procedure byptr(X: access T); -- primitiv für T

 type Ptr1 is access T;
 type Ptr2 is access T'class;
 procedure check2(X: Ptr2); -- primitiv für monomorphen Typ Ptr2
end A;
```

Anonyme Access Typen sind als Zuweisungsziel typkompatibel mit allen Access Typen gleichen Zieltyps T oder dessen Klassentyps T‘Class (aber nicht umgekehrt).



Anonyme Access Typen

```
package A is
 type T is tagged record …;
 procedure Check(X: T);
 procedure byptr(X: access T);
 type Ptr1 is access T;
 type Ptr2 is access T'class;
 procedure Check2(X: Ptr2);
end A;
```

```
package B is
 type NT is new A.T with …;
 procedure Check(X: NT);
 procedure byptr(X: access NT);
 type Ptr3 is access NT;
 type Ptr4 is access NT'class;
 procedure Check2(X: Ptr4);
end B;
```

```
Obj: B.NT;
Cobj: B.NT'class := B.Nt;
Dobj: A.T'class := B.Nt;
```

```
A.Check(Obj); -- illegal
B.Check(Obj); -- ok
A.Check(CObj); -- illegal !!
B.Check(DObj); -- illegal, wäre Downcast
A.Check(DObj); -- ok
B.Check(CObj); -- ok
```

Anonyme Access Typen

```
package A is
 type T is tagged record …;
 procedure Check(X: T);
 procedure byptr(X: access T);
 type Ptr1 is access T;
 type Ptr2 is access T'class;
 procedure Check2(X: Ptr2);
end A;
```

```
package B is
 type T is new A.T with …;
 procedure Check(X: NT);
 procedure byptr(X: access NT);
 type Ptr3 is access NT;
 type Ptr4 is access NT'class;
 procedure Check2(X: Ptr4);
end B;
```

```
Obj: B.NT;
Cobj: B.NT'class := B.Nt;
Dobj: A.T'class := B.Nt;
P3: B.Ptr3;
P4: B.Ptr4;
P2: A.Ptr2;
```

```
A.byptr(P3); -- illegal (Typfehler)
B.byptr(P3); -- ok
A.byptr(P4); -- illegal !!
B.byptr(P4); -- ok
A.byptr(P2); -- ok
B.byptr(P2); -- illegal, wäre Downcast
```

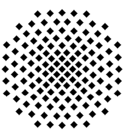
Anonyme Access Typen

```
package A is
 type T is tagged record …;
 procedure Check(X: T);
 procedure byptr(X: access T);
 type Ptr1 is access T;
 type Ptr2 is access T'class;
 procedure Check2(X: Ptr2);
end A;
```

```
package B is
 type T is new A.T with …;
 procedure Check(X: NT);
 procedure byptr(X: access NT);
 type Ptr3 is access NT;
 type Ptr4 is access NT'class;
 procedure Check2(X: Ptr4);
end B;
```

```
Obj: B.NT;
Cobj: B.NT'class := B.Nt;
Dobj: A.T'class := B.Nt;
P3: B.Ptr3;
P4: B.Ptr4;
P2: A.Ptr2;
```

```
A.Check2(P4); -- illegal (Typfehler)
A.Check(P3.all); -- illegal (Typfehler)
A.Check(P4.all); -- illegal
A.Check(P2.all); -- ok. Dispatching
```



Anonyme Access Typen

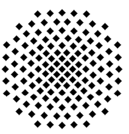
```
package A is
  type T is tagged record …;
  procedure Check(X: T);
  procedure byptr(X: access T);
  type Ptr1 is access T;
  type Ptr2 is access T'class;
  procedure Check2(X: Ptr2);
end A;
```

```
package B is
  type T is new A.T with …;
  procedure Check(X: NT);
  procedure byptr(X: access NT);
  type Ptr3 is access NT;
  type Ptr4 is access NT'class;
  procedure Check2(X: Ptr4);
end B;
```

```
Obj: B.NT;
Cobj: B.NT'class := B.Nt;
Dobj: A.T'class := B.Nt;
P3: B.Ptr3;
P4: B.Ptr4;
P2: A.Ptr2;
```

```
use A, B;
Check(Obj); -- ok, direct call
Check(CObj); -- dispatching
Check(DObj); -- dispatching
byptr(P4); -- dispatching
byptr(P2); -- dispatching
byptr(P3); -- ok, direkt
```

```
use A,B;
Check2(P2); -- ok, direct call
Check2(P3); -- ok, direct call
Check(P4.All); -- dispatching
Check(P2.All); -- dispatching
Check(P3.All); -- ok, direkt
```



Objektnotation (Ada 2005)

```
package A is
 type T is tagged record …;
 procedure Check(X: T);
 procedure byptr(X: access T);
 type Ptr1 is access T;
 type Ptr2 is access T'class;
 procedure Check2(X: Ptr2);
end A;
```

```
package B is
 type T is new A.T with …;
 procedure Check(X: NT);
 procedure byptr(X: access NT);
 type Ptr2 is access NT;
 type Ptr4 is access NT'class;
 procedure Check2(X: Ptr4);
end B;
```

```
Obj: B.NT;
Cobj: B.NT'class := B.Nt;
Dobj: A.T'class := B.Nt;
P3: B.Ptr3;
P4: B.Ptr4;
P2: A.Ptr2;
```

```
Obj.Check; -- ok
 CObj.Check; -- ok
 DObj.Check; -- ok
 P1.byptr; -- ok
 P2.byptr; -- ok
 P3.byptr; -- ok
 P2.Check2; -- kein tagged Type: keine Objektnotation
```

```
P1.All.Check; -- ok; direct call
P2.All.Check; -- dispatching
P3.All.Check; -- ok, direkt
P1.Check; -- ok, direct call
P2.Check; -- ok. Dispatching.
P3.Check; -- ok. direct call
```

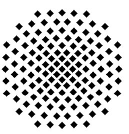
Objektnotation (Ada 2005)

Regeln:

- nur für Objekte vom tagged Typ T oder vom Typ T‘class

- nur für nicht mit Komponenten überlappende Methodennamen

- macht alle im gleichen Paket wie der Typ T deklarierten Unterprogramme durch die Selektion sichtbar, die T, T‘Class, access T oder access T‘Class als Typ des ersten Parameters haben (der dann im Aufruf durch das Präfix-Objekt gestellt wird).

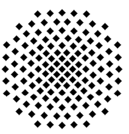


Anonyme Access Typen

```
package A is
  type T;
  type Ptr1 is access T;
  type Ptr2 is access T'class;
  type T is tagged record
     Next: Ptr2;
  end record;
 procedure SetNext(O: in out T; X: access T'Class);
end A;
   procedure SetNext(O: in out T; X: access T'Class) is
     begin O.Next := X; end SetNext;  -- illegal; Typfehler
     begin O.Next := X.all'access; end SetNext;  -- benötigt „all" bei Ptr2
     begin O.Next := Ptr2(X); end SetNext;       -- benötigt „all" bei Ptr2
```

Anonyme Access Typen (2005)

```
package A is
  type T;
 type Ptr1 is access T;
  type Ptr2 is access T'class;
  type T is tagged record
      Next: access T'class;  -- jetzt auch für Komponenten
  end record;
 procedure SetNext(O: in out T; X: access T‘Class);
end A;


procedure SetNext(O: in out T; X: access T‘Class) is
  begin O.Next := X; end SetNext;  -- ok
```



Aliased, All, Accessibility

```
package A is
  type T;
  type Ptr1 is access T;
  type Ptr2 is access all T'class;
  type T is tagged record
      Next: access T'class;
      Next1: Ptr2;
  end record;
  procedure SetNext(O: in out T;
                    X: access T‘Class);
end A;
```

```
Cobj : A.T‘class := … ;

begin
  declare
    Lobj: aliased A.T‘class := Cobj;
  begin
    Cobj.SetNext(Lobj'access);
```

***aliased**, weil eine Referenz durch* ***'access** entsteht*

***all**, weil* *X* *nicht notwendigerweise auf ein Heapobjekt zeigt*

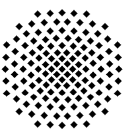
```
procedure SetNext(O: in out T; X: access T‘Class) is
  begin O.Next1 := Ptr2(X); end SetNext; -- Zuweisung verursacht hier Ausnahme
```

Overloading oder Overriding ? (Programmsicherheit)

Das Problem (1):

type T **is tagged ...;**
procedure Finalize(X: T);

type NT **is new** T;
procedure Finalise(X: NT); **-- keine Redefinion !**
procedure M(X:NT);

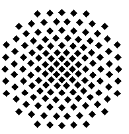


Overloading oder Overriding ? (Programmsicherheit)

Das Problem (2):

type T **is tagged ...;**
procedure Finalize(X: T);
procedure M(X: T); -- durch Wartung eingefügt

type NT **is new** T;
procedure Finalise(X: NT); **-- keine Redefinion !**
procedure M(X:NT); **-- wird heimlich zur Redefinition !**

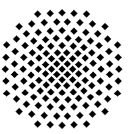


Overloading oder Overriding ?
(Programmsicherheit)

Das Problem (3):

type T **is new** OT ... ;
procedure M(X: T; Y: T);

type NT **is new** T …;
procedure M(X:NT; Y: NT);



Overloading oder Overriding ? (Programmsicherheit)

Das Problem (3):

type T **is new** OT ... ;
procedure M(X: T; Y: OT); -- durch Wartung geändert

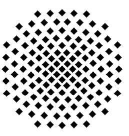
type NT **is new** T … ;
procedure M(X:NT; Y: NT); **-- ist keine Redefinition mehr !**

Schutz vor unbeabsichtigem Überladen oder Redefinieren

1. **overriding procedure** M(X: T); -- M muss eine Redefinition sein

2. **not overriding procedure** A(X:T); -- A muss eine neue Operation sein

3. **procedure** B(X:T); -- B darf redefinieren oder überladen

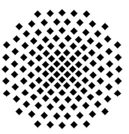


Verwendung der Vererbung in OOPLs

"If all you have is a hammer, everything looks like a nail."

oder

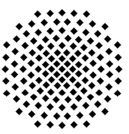
"Your unification is my obfuscation."



"is-a"-Beziehung

Die einzige Beziehung, die (fast) bedenkenlos durch Vererbung modelliert werden sollte. Für sie wurde das OO-Modell eigentlich geschaffen.

(Das "fast" bezieht sich auf das Ererben aus Bibliotheken, das in fast allen OOP-Sprachen Wartungsschwierigkeiten verursacht.)



"uses/needs"-Beziehung

In OOP-Sprachen, die ein Modul- oder Paketkonzept nicht kennen, wird Vererbung zur Realisierung dieser Beziehung (im wesentlichen zur Importierung) missbraucht.

In Ada wird dafür die Kontextklausel ("with", ggf. ergänzt mit "use") verwendet.

"is-implemented-by" Beziehung

Die "Standardlösung"

type Mine **is new** PKG.Implementing_Type;

ist nicht ideal, weil sie "is-implemented-by" mit "is-a" vermischt.

Sauberer via Interfaces oder mit:

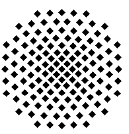
```
        type Mine is tagged private;
        procedure XYZ(X: Mine);
    private
        type Mine is new PKG.Implementing_Type;
        procedure XYZ(X: Mine) renames ZYX;
```

"has-a" Beziehung

Eine "has-a"-Beziehung sollte **NIE** durch Vererbung realisiert werden. Statt dessen werden (evtl. private) Komponenten für die "has-a" Beziehung zu einem Component_Type eingeführt...

```
        type Mine is new Parent with private;
private
        type Mine is new Parent with record
                                          X: Component_Type;
                                      end record;
```

... und Operationen auf "Mine" spezifiziert oder redefiniert, die durch Operationen auf der X-Komponente implementiert werden.



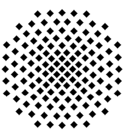
"Mixins"

- Erweiterung eines Typs durch vorgegebene Methoden(implementierungen) und Datenkomponenten (eines der Hauptargumente für Mehrfachvererbung)
- in Ada 95 über Generics realisiert

einfache generische "Mixins"

Wenn Datenkomponenten und Methoden sich eignen, von mehreren Klassen wiederverwendet zu werden, ohne ihrerseits eine sinnvolle Klasse mit eigenständigen Instanzen zu ergeben, dann lohnt sich ihre Spezifikation als generischer Mixin ...

```
generic
      type T is abstract tagged private;
   package List_Support is
      type NT is abstract new T with private;
      procedure append(To, New: in out NT);
      procedure remove(Elem: in out NT);
   private
      type NT is new T with record ...
         -- Datenkomponenten für Listenanschluss
   end List_Support;
```



einfache generische "Mixins"

- **"Beimengung" der Eigenschaften erfolgt durch Instantiierung:**

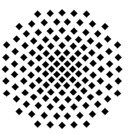
type XT **is tagged ...**;

package XT_Lists **is new** List_Support(XT);

type XT_listed **is new** XT_Lists.NT **with null record;**
    -- append und remove nun auf XT_listed Objekte
    -- anwendbar

alternativ:

subtype XT_listed **is** XT_Lists.NT; *-- netter Name; abstrakter Typ*
use type XT_Lists.NT; *-- Sichtbarkeit der prim. Operationen*



Eingeschränkte Generizität

Die Spezifikation des generischen Typparameter gibt die innerhalb der generischen Einheit verwendbaren Eigenschaften vor:

z.B.: **type** T **is limited private**; -- keine Operationen verfügbar
und viele Abstufungen bereits in Ada83 (siehe Reference Manual)

Für OOP, eingeschränkt polymorphe Parameter und Templates:

type T **is new** TP **with private**;
-- generische Einheit ist nur auf Typen anwendbar, die von
-- TP (transitiv) abgeleitet sind. Sie kann Eigenschaften
-- von TP im Umgang mit Objekten vom Typ T verwenden.

with package P **is new** GP(...); -- siehe später

Schwächen generischer Mixins

- "neue" Methoden können unbeabsichtigt existierende Methoden des aktuellen Typs redefinieren. In der Wartung hinzugefügte Methoden können von abgeleiteten Typen unbeabsichtigt redefiniert werden. (Ein generelles Problem aller OOPLs mit Mehrfach-vererbung oder anderen Mixin-Strategien, wenn Neu- und Redefinition nicht syntaktisch voneinander unterschieden werden.)

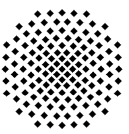
- Um auch auf abstrakte Typen anwendbar zu sein, muß sich der in der generischen Einheit deklarierte Typ als abstrakt bezeichnen. Der Instantiierer muß "konkretisieren", auch in Abwesenheit irgend-welcher abstrakter Operationen.

"Mixin" mit Access-Diskriminanten

Die Konstruktion auf den vorangegangenen Folien impliziert, daß die Operationen des Mixins zwar Zugriff auf primitive Operationen des zu erweiternden Typs im Rahmen spezifizierter generischer formalen Parameter besitzen, die umschließende **Objektinstanz** aber nicht kennen.

Wenn letztere Kenntnis notwendig ist, wird dies durch Access-Diskriminanten erreicht...

Wermutstropfen: Access_Diskriminanten nur für limitierte Typen !



Controlled Types

zwei vordefinierte "tagged" Typen im Paket Ada.Finalization:

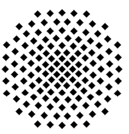
type Controlled **is abstract tagged private**;
-- folgende primitive Operationen werden implizit aufgerufen ...
procedure Initialize(Object: **in out** Controlled); *-- bei Default-Initialisierung*
procedure Adjust(Object: **in out** Controlled); *-- nach Kopieren*
procedure Finalize(Object: **in out** Controlled); *-- bei Ende der Lebensdauer*

type Limited_Controlled **is abstract tagged limited private**;

procedure Initialize(Object: **in out** Limited_Controlled);
procedure Finalize(Object: **in out** Limited_Controlled);

"Mixin" (2), generisch

```
-- Nachträgliches Einfügen der “Controlled” Eigenschaft
with Ada.Finalization;
generic
   type T is tagged limited private;
   with procedure Finalize(Obj: in out T'Class);
package Late is
    type Nt is new T with private;        -- T must be a limited type
private
    type Inner(D: access Nt'Class) is new
           Ada.Finalization.Limited_Controlled with null record;
    procedure Finalize(Obj: in out Inner);   -- is begin Finalize(Obj.D.all); end;
    type Nt is new T with record
       Anchor: Inner(D => Nt'access);
    end record;
end Late;
```



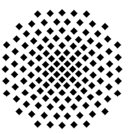
Ein kleiner "Hickup"...

warum nicht so ? :

```
generic
   type T is tagged limited private;
   with procedure Finalize(Obj: in out T);  -- statt T'Class
```

Weil dieses Finalize nicht dispatchen würde und auch nicht auf den im Kommentar postulierten Aufruf passt. D.h., der Instantiierer muß eine klassenweite Operation für den Parameter Finalize bereitstellen.

```
type T is ...;   -- tagged type
procedure Finalize(Obj: in out T); -- primitiv
...
procedure CW_Finalize(Obj: in out T'Class) is
          begin Finalize(Obj); end;
package Mixing_Controlled is new Late(T, CW_Finalize);
```



Zwei "is-a"-Beziehungen

Mit Einfachvererbung ist diese Situation schwierig zu modellieren.
Technisch geht es trotzdem...in der allgemeinsten Form:

```
        type P1_P2 is new Parent_1 with private;

        ... -- P2 Operationen und ggf. P1_P2 -> Parent_1 Konversionen
  private
        type P2_View (Self : access P1_P2'Class) is new Parent_2;
        type P1_P2 is new Parent_1 with record
                                      P2 : P2_View (Self  => P1_P2'Access);
                                    end record;
```

und Re-Export von Operationen von Parent_2 als P1_P2-Operationen,
implementiert durch P2-View Operationen auf der Komponente P2.

Einige Konsequenzen

- Implizite Kompatibilität von P1_P2 zu Parent_2 ist nicht gegeben.
- "Konversion" ist jedoch möglich über eine Funktion, die die P2 Komponente als "Inner" oder zu "Parent_2" konvertiert zurückgibt.
- Falls die Access-Diskriminante nötig ist, muß Parent_2 ein limitierter Typ sein. (Das ist eine echte Einschränkung !)

Anmerkung: der Mixin der "Controlled" Eigenschaft war eigentlich ein Beispiel für das Ererben entlang zweier "is-a" Beziehungen.

Mehrfachvererbung – das Problem

Mehrfachvererbung von Implementierungen (Daten oder Methoden) ist konzeptuell (und implementierungstechnisch) problematisch:

```
Class A
   datum X;
   method Y;
Class A1 extends A
   method Y;
Class A2 extends A
Class Dubious extends A1 and A2
Dubious Dub;
```

Worauf bezieht sich Dub.X oder Dub.Y ?

Es gibt jeweils zwei Möglichkeiten: A1.X oder A2.X; A1.Y oder A2.Y (= A.Y) ?

Mehrfachvererbung – die Lösung

Mehrfachvererbung von **Spezifikationen**, nicht aber von Implementierungen (Daten oder Methoden): Java Interfaces

```
interface A
  --- datum X; -- not allowed
  method Y;
interface A1 extends A
  --- method Y; -- unsinnig, weil A schon Y verspricht
interface A2 extends A
class Allright extends A1 and A2
      datum X;
      method Y = ….  -- Implementierung notwendig
Allright OK;
```

Worauf bezieht sich OK.X oder OK.Y ? Ist eindeutig und klar.

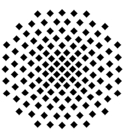
Mehrfachvererbung – Interfaces in Ada 2005

interface = abstract tagged null record;

ist das Prinzip. Der Rest folgt ☺ (fast).

```
type A is interface;
   procedure M(X: A) is abstract;
type A1 is interface and A;
  procedure N(X:A1) is abstract;
type A2 is interface and A;
      procedure M(X:A2) is abstract;
type Allright is new A1 and A2 with record
      X: Datum;
   end record;
   procedure M(X: Allright) is … -- Implementierung notwendig
   procedure N(X: Allright) is … -- Implementierung notwendig

OK: Allright;
OK1: A'Class := OK;
```

Worauf bezieht sich OK.X, OK.N, oder OK1.M ?   Ist eindeutig und klar.



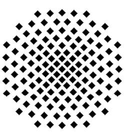
Null Procedures und Interfaces

Primitive Operationen von Interfaces haben keinen Rumpf und müssen daher entweder “is abstract” oder “is null” gesetzt werden.

Eine Procedure mit “is null” zu deklarieren ist neu in Ada 2005.

Diverse Verwendungszwecke für "is null":

- als Default am Ende der Redefinitionskette, so dass immer eine Vateroperation aufgerufen werden kann.
- als "Hook" für einen "call-out", der im Default-Fall dann nichts tut.
- semantisch äquivalent zu "is begin null; end;" aber als Teil der Spezifikation wichtige Vertragsinformation



Null Procedures und Interfaces

Null Prozeduren als primitive Operationen z.B. eines Controlled Types:

procedure Finalize(Obj : **in out** Controlled) **is null**;

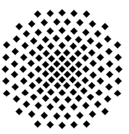
.. oder als Default für formale Prozeduren eines generischen Pakets, z.B.:

```
generic
   with procedure Pre_Action_Expr(E : Expr) is null;
   with procedure Post_Action_Expr(E : Expr) is null;
   with procedure Pre_Action_Decl(D : Decl) is null;
   …
package Tree_Walker is
```

"Limited" und "Synchronized" Interfaces

Das Interface Konzept wurde auf "protected" und "task" types ausgedehnt, wobei

- ein “limited” Interface implementiert/ererbt werden kann durch:
  - einen "limited" oder nicht-"limited" tagged Typ oder Interface
  - ein "synchronized" Interface
- ein “Synchronized” Interface implemented werden kann durch:
  - Task interfaces und Typen
  - "protected" Interfaces und Typen
- die bereits objekt-orientierte Präfix-Notation für Tasks und Protected Objects automatisch umgesetzt wird.



Beispiel eines Synchronized Interface

Ein protected Interface implementiert (ererbt) ein synchronized Interface:

```
type Buffer is synchronized interface;
procedure Put(Buf : in out Buffer;
          Item : in Element) is abstract;
procedure Get(Buf : in out Buffer;
          Item : out Element) is abstract;

protected type Mailbox(Capacity : Natural) is new Buffer with
   entry Put(Item : in Element);
   entry Get(Item : out Element);
private
   Box_State : …
end Mailbox;
```

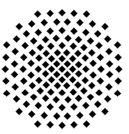
Synchronized Interfaces

Ein Task Interface implementiert (ererbt) ein synchronized Interface:

```
type Active_Buffer is task interface and Buffer;
procedure Put(Buf : in out Active_Buffer;
            Item : in Element) is abstract;
procedure Get(Buf : in out Active_Buffer;
            Item : out Element) is abstract;
procedure Set_Capacity(Buf : in out Active_Buffer;
                 Capacity : in Natural) is abstract;
```

Ein Task Typ implementiert ein Task Interface:

```
task type Postal_Agent is new Active_Buffer with
   entry Put(Item : in Element);
   entry Get(Item : out Element);
   entry Set_Capacity(Bag_Capacity : in Natural);
   entry Send_Home_Early;  -- An extra operation.
end Postal_Agent;
```

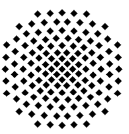


Zyklisch abhängige Pakete (oft mit tagged Typen)

```
with Department;
package Employee is
   type Object is tagged private;
   procedure Assign_Employee (Who            : in out Employee.Object;
                              To_Department  : in out Department.Object);
private
   type Object is tagged
      record
         Assigned_To : access Department.Object;
      end record;
end Employee;
```

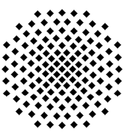
ILLEGAL !!

```
with Employee;
package Department is
   type Object is tagged private;
   procedure Choose_Manager (For_Department : in out Department.Object;
                             Who            : in out Employee.Object);
private
   type Object is tagged
      record
         Manager : access Employee.Object;
      end record;
end Department;
```



Zyklisch abhängige Pakete (oft mit tagged Typen)

```
package Department is
    type Object is tagged;          "as if"  implizit vorhanden
 end Department;


 limited  with Department;
 package Employee is
    type Object is tagged private;
    procedure Assign_Employee (Who             : in out Employee.Object;
                               To_Department   : in out Department.Object);
 private
    type Object is tagged
       record
          Assigned_To : access Department.Object;
       end record;
 end Employee;

 with Employee;
 package Department is
    type Object is tagged private;
    procedure Choose_Manager (For_Department : in out Department.Object;
                              Who            : in out Employee.Object);
 private
    type Object is tagged
       record
          Manager : access Employee.Object;
       end record;
 end Department;
```



Limited With Klausel

- Öffnet Sichtbarkeit auf einen *limited view* des Pakets
  - Enthält nur Typen und genestete Pakete (und deren Typen) des Pakets
  - Typen sind semantisch als unvollständige Typen anzusehen
  - Abhängigkeitszyklen, die durch limited with Klauseln entstehen, sind erlaubt
  - Bedingt eine gewisse Form von "Kurzbesuch" des Pakets
  - vervollständigt wird der Typ erst dort, wo reguläre With Klauseln auf beide involvierten Pakete gültig sind

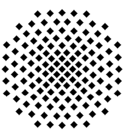
Formale generische Pakete

Weiterer generischer Parameter in Ada95:

with package P1 **is new** GP(<>);
-- alle Instantiierungen von GP als aktueller Parameter möglich
with package P2 **is new** GP(**P1, P2**);
-- nur Instantiierungen von GP mit zu P1 und P2 passenden
-- Parametern sind möglich

Wozu ?

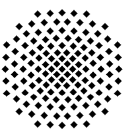
Übung für den Hörer: warum nur für generische Pakete ?



Gruppierung

```
package P is
   type T is ...
   procedure XYZ;
   type X is ...
end P;

generic   -- Ada83/95
    type FT is private;
    with procedure FP;
    type FX is private;
package Gen_Pkg is ...

package My_Pkg is
          new Gen_Pkg(T, XYZ, X);
```

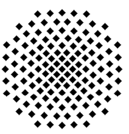
```
generic
package P is
   type T is ...
   procedure XYZ;
   type X is ...
end P;
package My_P is new P;

generic -- Ada95
    with package FP is new P(<>);
package Gen_Pkg is ...

package My_Pkg is
          new Gen_Pkg(My_P);
```



konsistente Gruppierung

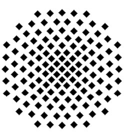
```
package P is
  type T is ...
  type X is record C:T; ...
end P;

generic   -- Ada83/95
   type FT is private;
   type FX is ??????; ☹
package Gen_Pkg is ...

package My_Pkg is
          new Gen_Pkg(T, X);
```

```
generic
package P is
  type T is ...
  type X is record C:T; ...
end P;
package My_P is new P;

generic -- Ada95
   with package FP is new P(<>);
package Gen_Pkg is ...

package My_Pkg is
          new Gen_Pkg(My_P);
```



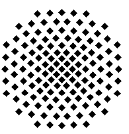
konsistente Instantiierungen

```
generic
   type FT is private;
package GP1 is
         type NT is ...;
         Obj: NT;
end GP1;
generic
   with package P1 is new GP1(<>);
   with package P2 is new GP2(P1.NT, P1.Obj);
         -- Beschränkungen in Abhängigkeit der Deklarationen in P1
   with package P3 is new GP3(P1.FT);
         -- ... oder in Abhängigkeit der aktuellen Parameter von P1
package new_Pkg is ...
```



"Signatur"-Pakete

```
generic
    type T is tagged private;
    procedure XYZ(X:T);
    with package P is new GP(T);
package Signature is end Signature;  -- mit Absicht leer

package My_Interface is new Signature(T1, Abc, P);
   -- Effekt: statische Prüfung, daß die Parameter dem generischen
   -- Vertrag entsprechen; außerdem kann My_Interface nun sorglos
   -- jedem Paket übergeben werden, das diese Signatur verlangt, z.B.
generic
   with package X is new Signature(<>);
package Y is  -- kann auf die aktuellen Parameter von X zugreifen
```



Signaturpakete = (Java) Interfaces ?

eine große Ähnlichkeit besteht:

- der "generische Vertrag" des Signaturpakets entspricht der Zusage bestimmte Dienste zu implementieren.
- das formale generische Paket entspricht dem Interface-View

- Durch das möglich werdende Renaming und das nachträgliche Zusammenfassen bestehender Dienste erscheint das Ada-Modell mächtiger.

- ein verbleibendes Problem: die bereits angesprochene Unfähigkeit der generischen Parameter, zwischen primitiven und nicht-primitiven Operationen zu unterscheiden.