Ada 95

Vererbung in Ada 95 lässt sich auf diese Kurzformel bringen:

Ableitung + Überschreiben + Typerweiterung
+ dynamische Typerkennung

Abgeleitete Typen "erben" Operationen:

```
type Int is range 0..10000;
function Plus (Links, Rechts: Int) return Int;
type Laenge is new Int;
```

Operationen abgeleiteter Typen sind überschreibbar:

```
type Winkel is new Int;
function Plus (Links, Rechts: Winkel) return Winkel;
```

Kontravarianz versus Kovarianz

Kovarianz: Parametertypen überschr. Op. sind Untertypen.
Kontravarianz: Parametertypen überschr. Op. sind Obertypen.
Invarianz: Parametertypen bleiben unverändert.

Das Ersetzbarkeitsprinzip verlangt, dass Eingangsparameterty-pen kontravariant (oder invariant) sind.

In Ada sind alle Parametertypen (auch Eingangsparameterty-pen), die bei der Typableitung verändert werden, kovariant.
Das Ersetzbarkeitsprinzip ist verletzt.

Daher ist es nicht möglich, eine Instanz eines Untertyps zu verwenden wo eine Instanz eines Obertyps erwartet wird.

In Ada müssen die Argumenttypen den Parametertypen (ausser für klassenweite Typen) genau entsprechen.

Typerweiterung

Typerweiterung ist das Hinzufügen neuer Felder zu abgeleiteten Verbundtypen.

Erweiterbare Verbundtypen sind mit `tagged` gekennzeichnet:

```
type Person is tagged record
    Vorname: String (1..20);
end record;

type Mann is new Person with record
    Barttraeger: Boolean;
end record;

type Frau is new Person with null record;
```

Typumwandlung ist für Instanzen erweiterbarer Verbundtypen nur in Richtung Obertyp möglich.

Klassenweite Typen

Zu jedem erweiterbaren Verbundtyp t ist implizit ein klassenweiter Typ t`'Class` definiert. Die durch t definierte Klasse (in Ada-Terminologie) umfasst t und alle von t abgeleiteten Typen.

Jede Instanz eines klassenweiten Typs enthält ein unsichtbares Feld ("tag") das den spezifischen Typ angibt.

Da das "tag" während der Programmausführung gebraucht wird, muss jede Variable eines klassenweiten Typs mit einer Instanz eines spezifischen Typs initialisiert werden.

Spezifische Typen von Variablen sind unveränderbar. Zeiger können auf Instanzen verschiedener spezifischer Typen zeigen.

Klassenweite Typen (Beispiel)

```
function Anrede (P: in Person) return String; -- ""
function Anrede (M: in Mann) return String; -- "Herr"
function Anrede (F: in Frau) return String; -- "Frau"

procedure Display (P: in Person'Class) is
begin
    Put (Anrede (P));
    Put (P.Vorname);
    if P in Mann'Class then
        Put (" (maennlich)")
    elsif P in Frau'Class then
        Put (" (weiblich)")
    end if;
end Display;
```

Dynamisches Binden erforderlich

Abstrakte Typen

Abstrakte Typen definieren klassenweite Typen, können aber nicht als spezifische Typen auftreten.

Abstrakte Typen werden als `abstract` gekennzeichnet:

```
type A is abstract tagged null record;
procedure Q (X: in A) is abstract;
type B is new A with record
    I: Integer;
end record;
procedure Q (X: in B) is ...;
```

Jeder abgeleitete konkrete Typ muss alle ererbten abstrakten Operationen überschreiben.

Einschränkungen auf Typparametern

```
generic
    type T is private;
    with function "<"(X,Y:T) return Boolean;
function Max(X,Y:T) return T is
begin
    if X < Y
        then return Y;
        else return X;
    end if;
end Max;

function MaxI is new Max(Integer,"<");
function MinF is new Max(Float,">");
```

Datenkapselung

Datenkapselung und Vererbung beruhen in Ada 95 auf getrennten Mechanismen: Datenkapselung auf Paketen und Vererbung auf abgeleiteten Typen.

In einigen anderen Sprachen (C++, Eiffel) sind diese Mechanismen zusammengefasst.

Welcher Ansatz ist überlegen?

- Für ein Zusammenfassen spricht, dass (abgeleitete) Typen abgeschlossene Einheiten bilden sollten.
- Dagegen spricht, dass für die Fälle, in denen mehrere Typen eng aneinander gekoppelt sind, komplizierte Mechanismen zur teilweisen Aufhebung der Datenkapselung notwendig werden (z.B. Friends in C++).

Beispiel für Zusicherungen in Eiffel

Eine Besonderheit von Eiffel sind Zusicherungen in Form von "preconditions", "postconditions" und Invarianten:

```
Einzahlen(Summe: T) is
    require Summe >= 0
    do Addieren(Summe)
    ensure Guthaben = old Guthaben + Summe
    end; -- Einzahlen

Abheben(Summe: T) is
    require Summe >= 0;
            Guthaben >= Summe - Ueberziehungsrahmen
    do Addieren(-Summe)
    ensure Guthaben = old Guthaben - Summe
    end; -- Abheben
```

Zusicherungen in Eiffel

Konzeptuell gehören Zusicherungen zur Schnittstelle.

Zusicherungen sind dem Aufrufer einer Funktion bekannt.

Zusicherungen werden, sofern möglich, vom Compiler bzw. Linker überprüft.

Wenn das nicht möglich ist, erfolgt die Überprüfung zur Laufzeit. Beim Erkennen einer unerfüllten Zusicherung wird eine Ausnahmebehandlung eingeleitet.

Vererbung in Eiffel

Eiffel unterstützt Vererbung von Klassen:

```
class SCHILLING_BETRAG inherit GELD_BETRAG feature ...
```

Wie in Ada 95 gibt es die Möglichkeit, Operationen zu überschreiben, Klassen (erweiterbare Verbunde) zu erweitern und abstrakte Typen zu definieren.

In Eiffel entsprechen alle Variablen ohne zusätzliche Deklarationen Zeigern auf klassenweite Typen in Ada 95.

Im Gegensatz zu Ada 95 bietet Eiffel Mehrfachvererbung.

Kovarianz und Kontravarianz in Eiffel

Eingabeparametertypen sind in Eiffel kovariant.
Es können Instanzen von Untertypen übergeben werden wo Instanzen von Obertypen erwartet werden.
Dadurch gibt es Probleme bei der statischen Typüberprüfung.

“Preconditions” sind kontravariant: Sie müssen schwächer sein als in Oberklassen.
Begründung: Bei Nichterfülltsein einer stärkeren “precondition” müsste eine Ausnahmebehandlung eingeleitet werden.

“Postconditions” sind kovariant: Sie sind stärker als die in Oberklassen.

Unterbereichstypen in Ada sind eine vereinfachte Form von Zusicherungen. Diese sind generell kovariant – das Ersetzbarkeitsprinzip ist daher nicht erfüllt.

Variante Parametertypen (1)

Prinzip der Ersetzbarkeit: Eine Instanz eines Untertyps ist erlaubt wo eine Instanz eines Obertyps erwartet wird.

Es gilt entsprechend: $s \rightarrow t \leq s' \rightarrow t'$ wenn $s' \leq s$ und $t \leq t'$.

Partielle Funktionen als Alternative: Jeder Typ beschreibt nur einen Teil einer Gesamtfunktion. Argumenttypen bestimmen, welche Einzelfunktion auszuführen ist.

Im allgemeinen *"multiple dispatch"*. Wenn jede Einzelfunktion nur von einem Argumenttyp abhängt: *"single dispatch"*.

"Single dispatch" möglich wenn die Typen aller variierenden Eingangsparameter in Einzelfunktionen stets gleich sind (Ada). Dies erfordert jedoch eine Laufzeit-Überprüfung.

Variante Parametertypen (2)

Ansatz in funktionalen Sprachen: kontravariante Eingangs-, kovariante Ausgangs-, invariante Durchgangsparameter; statische Typüberprüfung lokal durchführbar.

Ansatz partieller Funktionen: alle Parameter kovariant; statische Typüberprüfung in modernen Systemen bei "multiple dispatch" durchführbar (nicht lokal).

Kombination dieser beiden Ansätze möglich: Man teilt die Parameter entsprechend den Ansätzen in zwei Klassen.

Kovariante Eingangsparametertypen sind erwünscht.

Nicht möglich: lokal — stark — uneingeschränkt — kovariante Eingangsparameter.

“Subtyping” versus Vererbung

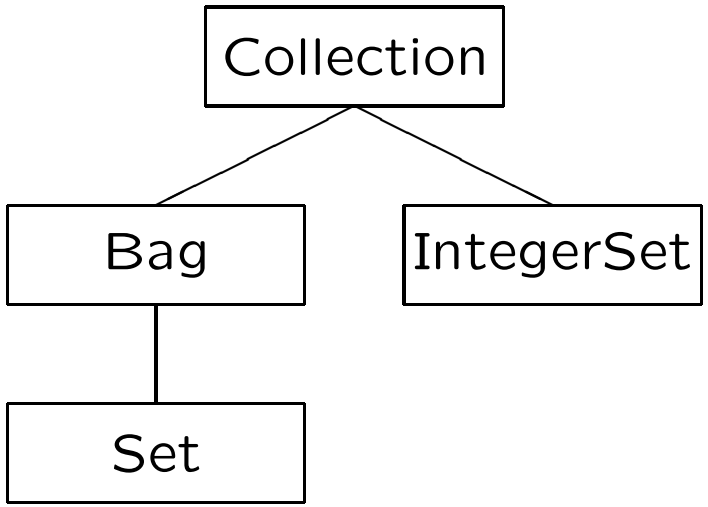
Bisherige implizite Annahme: Vererbung = “subtyping”.

In objektorientierten Sprachen beschreibt ein Typ *das nach aussen sichtbare Verhalten eines Objekts.*

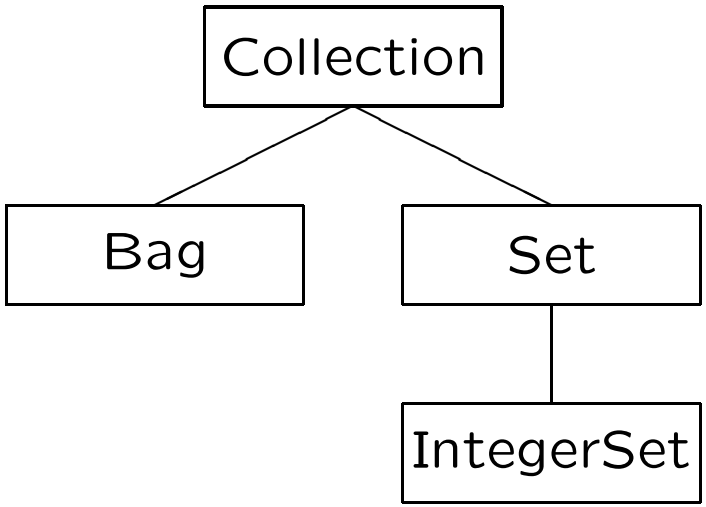
Vererbung entspricht dem Wiederverwenden von Code.

Beschreibung des Verhaltens und Implementierung müssen verschieden sein, da sonst das Überschreiben von Operationen verboten wäre.

Beispiel: “Subtyping” $\neq$ Vererbung



Vererbung von Impl.



Vererbung von Konzepten

Trennung Typ / Implementierung

Beispiel in Portlandish:

```
type Student
    supertypes Person
    signature getMnr (): Integer
endType

implementation StudentImpl of Student
    superImplementations PersonImpl
    fields mnr: Integer
    methods getMnr () { return mnr }
endImplementation
```

Spezifikation des Verhaltens

Weder Schnittstellen noch Implementierungen sind zur Spezifikation des sichtbaren Verhaltens von Objekten geeignet.

Stand der Technik: formale oder informale Beschreibungen in der Dokumentation.

Ziel: Der Compiler überprüft formale Spezifikationen eines Typs automatisch auf Typintegrität.

Ein erster Ansatz: Zusicherungen in Eiffel.

Aktuelle Forschung: Darstellung und Überprüfung komplexerer Spezifikationen, die auf logischen und algebraischen Methoden beruhen.

Beispiel: Typspezifikation in Larch

bag = **type**
uses BBag (bag **for** B)
for all b: bag
put = **proc** (i: int)
requires $|b_{pre}.elems| < b_{pre}.bound$
modifies b
ensures $b_{post}.elems = b_{pre}.elems \cup \{i\}$
$\wedge\ b_{post}.bound = b_{pre}.bound$

get = **proc** () **returns** (int)
requires $b_{pre}.elems \neq \{\}$
modifies b
ensures $b_{post}.elems = b_{pre}.elems - \{result\}$
$\wedge\ result \in b_{pre}.elems \wedge b_{post}.bound = b_{pre}.bound$

Konsistente Definition von “subtyping”

Ein Typ s ist ein Untertyp von t wenn

- jede Invariante von s eine Invariante von t impliziert;
- die Methoden von s sich wie die Methoden von t verhalten:
  - Typen von Eingangsparametern kontravariant sind;
  - Ergebnistypen kovariant sind;
  - die Menge der von s erlaubten Ausnahmebehandlungen in jener von t liegt;
  - “preconditions” von t die entsprechenden “preconditions” von s implizieren;
  - “postconditions” von s die entsprechenden “postconditions” von t implizieren;
- “constraints” auf s “constraints” auf t implizieren.

Untertypen — Generizität (Beispiel)

```
generic
  type Data is private;
package Lists is
  type E is private;
  function App(A,B: access E)
              return access E;
private
  type E is record
    Item: Data;
    Next: access E
  end record;
end Lists;
type T is ...
package TLists is new Lists(T);
```

```
type E is tagged record
  Next: access E
end record;
function App(A,B: access E)
            return access E;
type TE is new E with ...
```

Untertypen versus Generizität

Subtyping und Generizität sind manchmal gegeneinander austauschbar, aber nicht immer.

- Subtyping ist notwendig für heterogene Datenstrukturen.
- Generizität erlaubt oft, Einschränkungen der Varianz von Parametern zu umgehen.

Gebundene Typparameter kombinieren Subtyping und Generizität (z.B. `KONTO[T->GELD_BETRAG]` in Eiffel).

Subtyping ist auch auf generischen Typen möglich, aber im allgemeinen kompliziert. Z.B. in Eiffel (nur Kovarianz):

wenn	`DOLLAR_BETRAG`	$\leq$	`GELD_BETRAG`
dann	`KONTO[T->DOLLAR_BETRAG]`	$\leq$	`KONTO[T->GELD_BETRAG]`

Implizite Untertypbeziehungen

In Java und C#:

`Student[]` Untertyp von `Person[]`
wenn `Student` Untertyp von `Person`

Aber: in starkem Typsystem NICHT typsicher

Dynamische Überprüfung und Exception wenn Instanz von `Person` in Instanz von `Student[]` geschrieben

Daher: `List<Student>` kein Untertyp von `List<Person>`
und `List<Person>` kein Untertyp von `List<Student>`

(keine impliziten Untertypbeziehungen)

Gebundene Generizität, Java-Wildcards

```
class List<T extends Comparable<T>> {
    public void add(T x) { ... if (x.equal(y)) ... }
    public void copyFrom (List<? extends T> f) {...}
    public void copyTo (List<? super T> t) {...}
}
interface Comparable<T> { boolean equal (T that); }
class Integer extends Comparable<Integer> {
    public boolean equal (T that) {...}
}
class MyInt extends Integer {...}
...  List<MyInt> lm; List<Integer> li;
     lm.copyTo(li); li.copyFrom(lm);  ...
```

Funktionen über Typen

F-gebundener Polymorphismus (wie in Java): $S \leq T[S]$

Subtyping höherer Ordnung: $S <\# T$ wenn $(\forall U)\, S[U] \leq T[U]$

Beispiel: $\{\texttt{i}{:}\texttt{int}, \texttt{f}{:}u{\to}u, \texttt{next}{:}u\} <\# \{\texttt{i}{:}\texttt{int}, \texttt{f}{:}v{\to}v\}$

Für gebundene Generizität verwendbar: $U[x <\# T]$

Es wird versucht, Subtyping durch Matching ($<\#$) zu ersetzen, obwohl Matching das Ersetzbarkeitsprinzip nicht erfüllt.

ψ-Terme in LIFE

ψ-Terme kombinieren logische Terme mit Typen, auf denen ein Verband definiert ist:

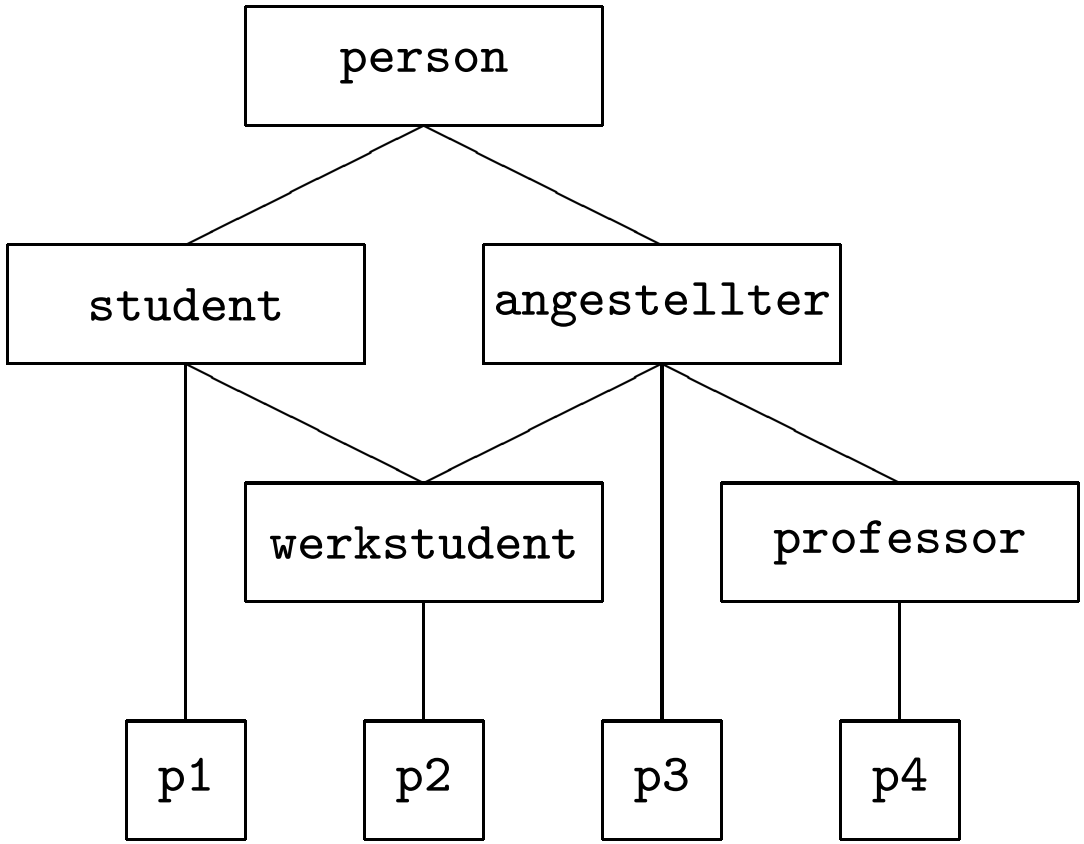
```
X:person(name => nameT(first => string,
                       last => S:string),
         married => person(name => nameT(last => S),
                           married => X))
```

`person`, `nameT` und `string` sind *Sorten* (Typsymbole).

`name`, `first`, `last` und `married` sind Label zur Bezeichnung von *Attributen.*

`X` und `S` sind *Variablennamen.*

Beispiel: Verband über Sorten



Unifikation von ψ-Termen

Die Unifikation von

```
student(betreuer => professor(assistent => angestellter),
        freund => person)
```

mit

```
X:person(betreuer => p4(assistent => X),
         vermieter => person)
```

liefert als Ergebnis

```
X:werkstudent(betreuer => p4(assistent => X),
              freund => person,
              vermieter => person)
```

LIFE-Programme

LIFE-Programme ähneln Prolog-Programmen.

Programmstück:

```
prueft(p4, p1).

fleissig(angestellter).
fleissig(X:student) :- prueft(professor, X).
```

Die Anfrage "`fleissig(Y:student)`" liefert nacheinander die beiden Antworten "`Y=werkstudent`" und "`Y=p1`".

Beispiel: “task types” in Ada

```
task type Buffer is
    entry Put (E: in Element);
    entry Get (E: out Element);
end Buffer;

task body Buffer is
    X: Element;
begin
    loop
        accept Put (E: in Element) do X := E; end;
        accept Get (E: out Element) do E := X; end;
    end loop;
end Buffer;
```

Beispiel: Algebra für Prozesstypen

```
 Buffer = Put (in Element) *
          Get (out Element) *
          Buffer

IBuffer = ( Put (in Element) *
            Get (out Element) ) ||
          IBuffer

UBuffer = Put (in Element) ||
          Get (out Element) ||
          UBuffer
```

“Subtyping” für Prozesstypen

Die Untertyp-Relation $\leq$ wird durch Regeln definiert:

$$\tau \leq \tau \qquad \tau \leq \varepsilon \text{ (ε entspricht dem leeren Typ $\top$)}$$

$$\frac{\tau_1' \leq \tau_1 \ \ldots \ \tau_n' \leq \tau_n}{m(\tau_1, \ldots, \tau_n) \ \leq \ m(\tau_1', \ldots, \tau_n')} \qquad \frac{\tau' \leq \tau''}{\tau * \tau' \ \leq \ \tau * \tau''}$$

$$\frac{\tau \leq \tau'' \quad \tau' \leq \tau'''}{\tau \mid\mid \tau' \ \leq \ \tau'' * \tau'''} \qquad \frac{\tau \leq \tau'' \quad \tau' \leq \tau'''}{\tau \mid\mid \tau' \ \leq \ \tau'' \mid\mid \tau'''} \qquad \frac{\tau \leq \tau'' \quad \tau' \leq \tau'''}{\tau + \tau' \ \leq \ \tau'' + \tau'''}$$

Typüberprüfung für Prozesstypen

Der Prozesstyp einer Variablen verbrieft das *Recht*, entsprechende Nachrichten zu senden.

Prozesstypen von Variablen ändern sich wenn Nachrichten an durch die Variablen referenzierten Prozesse geschickt werden.

Bei einer Zuweisung $x := y$ wird der ursprüngliche Prozesstyp $\tau \mid\mid \tau'$ von y so aufgeteilt, dass x den Prozesstyp τ und y den Prozesstyp τ' erhält.

Alle indeterministischen Alternativen haben denselben Typ.

Diese Prozesstypen erlauben statische Typüberprüfungen.

Vererbungsanomalie

Vererbung von Implementierungen in nebenläufigen Sprachen, vor allem das Hinzufügen neuer Operationen, ist schwierig.

Grund: Synchronisationsbedingungen ändern sich.

- Synchronisationscode getrennt von Implementierung.

  Problem: In der Praxis oft nicht durchführbar.

- Umkehrung der Vererbungshierarchie.

  Problem: Erweiterungen der Klassen nicht möglich.

- Synchronisationsbedingungen dürfen sich nicht ändern (entsprechend Ersetzbarkeitsprinzip wie bei Prozesstypen).

  Problem: Technik noch nicht ausgereift.

Smalltalk

Variablen sind nicht typisiert.

Jede Nachricht kann an jedes beliebige Objekt geschickt werden. Eine Methode ist für unverstandene Nachrichten vorgesehen.

Alle Typüberprüfungen erfolgen zur Laufzeit.

Subtyping und Generizität sind nicht notwendig.

Klassen und Einfachvererbung werden unterstützt.

C++

Typen, Module und Verbunde zu Klassen vereint.

Virtuelle und nichtvirtuelle Funktionen.

Sichtbarkeit durch Schlüsselwörter festlegbar.

Lockerung der Sichtbarkeitsregeln durch „friends“.

Umfangreiches Typkonzept mit invarianter Mehrfachvererbung, Generizität, abstrakten Typen, Überladen und RTTI. Alle Typüberprüfungen statisch.

Löcher durch explizite Typumwandlungen, nicht überprüfte Feld- und Variantenzugriffe und Probleme bei dynamischer Speicherverwaltung.

Java und C#

Löcher im Typsystem von C++ geschlossen.

Typüberprüfungen zum Teil dynamisch.

Teilweise Trennung von Schnittstellen und Klassen.
Mehrfachvererbung nur für Schnittstellen.

Generizität hat sich noch nicht praktisch durchgesetzt.
C#: Unterstützung von Generizität unklar.

Java: schwaches Modulkonzept.

Ada 95

Sehr umfangreiches Typsystem.

Trennung zwischen Modulen und Typen/Verbunden.

Vererbung teilweise kovariant (keine Ersetzbarkeit).

Typüberprüfungen teilweise dynamisch.

Bereichseinschränkungen.

Ausschweifende Syntax.

Eiffel

Typen, Module und Verbunde zu Klassen vereint.

Gezieltes Sichtbarmachen von „features“ in anderen Klassen.

Umfangreiche Möglichkeiten zum Umbenennen und Überschreiben von „features“.

Umfangreiches Typsystem mit kovarianter Mehrfachvererbung, abstrakten Typen, gebundener Generizität, RTTI.

„Design by contract“ durch Zusicherungen.

Typüberprüfungen teilweise dynamisch.

Ausschweifende Syntax.