# Absztrakt adattípusok

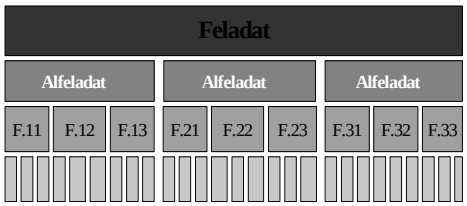
**Procedurális- vagy adatabsztrakció?**

Egy szoftver rendszer mindig végrehajt bizonyos **tevékenységeket bizonyos** a**datokon**.

- A tervezés központi kérdése, hogy a rendszer felépítését mire alapozzuk:
  - –a **végrehajtandó tevékenységekre**, **vagy**
  - –az **adatokra**?

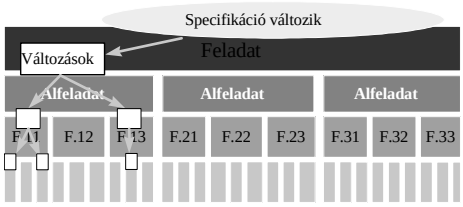
- **Procedurális absztrakció**

**Top-down megoldás**




A megközelítés hátránya: ha változik a specifikáció.

Mivel egy alprogram specifikáció a magasabb szintű specifikációtól függ, így a változás tovább gyűrűzik lefelé, és végül egy egész kis változásnak nagyon nagy hatása (és költsége) lehet.

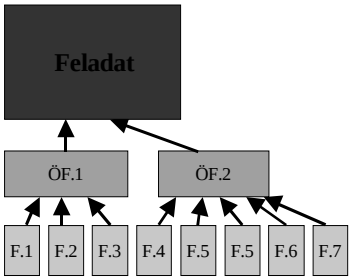



**A** specifikáció változásának hatása

## Adatabsztrakció

- Ha elég magas szintről analizáljuk a problémát, úgy tűnik, a rendszerek jobban jellemezhetőek hosszú távon objektumaikkal, mint a rájuk alkalmazott tevékenységekkel.
- Ezt a megközelítést választva, először megpróbáljuk jellemezni az objektumokat, és az objektumok osztályait (az adattípusokat) amelyek szerepet játszanak a rendszerben. Az új adattípusokat a már meglévő felhasználásával tervezzük – ez a bottom-up tervezés. Ez azt jelenti, hogy az elérhető komponensek szintjéről indulunk, abból építkezünk.
- Ezután oldjuk meg a problémát.

## Alulról felfelé építkezés



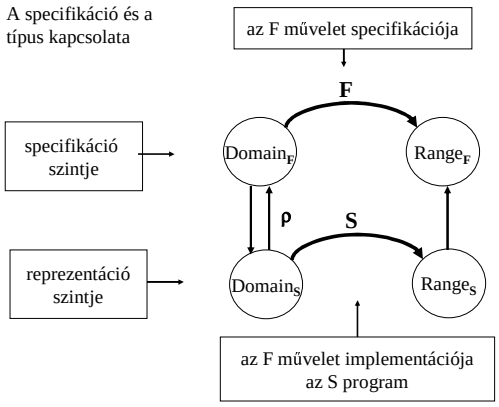
## Típus a fordító program szemszögéből

- **Egy absztrakció, ami összefoglalja egy programban előforduló bizonyos objektumok bizonyos közös tulajdonságait.**
- **Ezek a tulajdonságok:**
  - **a kódolás (encoding),**
  - **a méret (size),**
  - **a szerkezet (structure),**
  - **magasabb szinten a szemantika (semantics).**
- **A típus jelentését a típusérték-halmaz és a típusműveletek definiálásával adhatjuk meg.**

## Típus a programozó szemszögéből

- Típusspecifikáció („mit”)
  - alaphalmaz: a valós világ minket érdeklő objektumainak halmaza
  - specifikációs invariáns ($I_S$) - ezt a halmazt tovább szűkíti
  - típusműveletek specifikációja

- Típus megvalósítás („hogyan”)
  - Reprezentációs függvény
  - Típus invariáns
  - Típusműveletek megvalósítás




## Elvárások és eszközök

- Elvárások a programozási nyelvekkel szemben
- Absztrakt típusok megvalósításának *eszközei*

## Elvárások a programozási nyelvekkel szemben

- **Modularitás**
  - az egyes típusokat önálló fordítási egységekben lehessen megvalósítani. Ez biztosítja a típusok újrafelhasználhatóságát valamint a hatékony programfejlesztést, hiszen az egyes modulok könnyedén átvihetők más programokba, és a különböző egységeket más-más programozó is fejlesztheti, nem zavarva egymás munkáját.

- **Adatrejtés**
  - a nyelv támogassa a reprezentáció elrejtését. Ezen eszköz segítségével a nyelv maga biztosítja, hogy az adott típus használója csak a specifikációban megadott tulajdonságokat használhassa ki. Ez a megszorítás lehetővé teszi a reprezentáció, illetve az implementáció megváltoztatását anélkül, hogy a változások a programban felfelé gyűrűznének.

- **Specifikáció és implementáció szétválasztása külön fordítási egységbe**
  - Ekkor az adott típust használó más modulok a típus specifikáció birtokában elkészíthetők, függetlenül a tényleges implementációtól.

- **Modulfüggőség kezelése**
  - A fordító program kezelje maga a modulok közötti relációkat (egyik használja a másikat stb.).

- **Konzisztens használhatóság**
  - A felhasználói és a beépített típusok ne különbözzenek a használat szempontjából! A típusokat minél inkább a valós világban megszokott, "természetes" módon lehessen kezelni!

- **Generikus programsémák támogatása**.
  - A programozó lehetőleg minél általánosabban írhassa meg programjait. A nyelv adjon lehetőséget az ismétlések minimalizálására, nem csak a közvetlen kódismétlések elkerülésére, hanem a magasabb szintű megoldási struktúrák többszörös megírásának elkerülésére. Ez nagyban javítja a kód olvashatóságát és karbantarthatóságát.

## Absztrakt típusok megvalósításának *eszközei*

- **Modulokra bontás**

A professzionális használatra szánt programozási nyelvek mindegyikében megjelenik a modularizáció támogatása. A modernebb nyelvekben a modularizáció alapját egyre inkább a típusokra bontás jelenti, azaz egy modul egy típust implementál.
Teljesen ezen az alapon csak kevés nyelv -
- sokszor van szükség "alprogram könyvtárak"-ra.
- szükség lehet csak implementációs célokat szolgáló típusok megvalósítására is, amelyeket ilyenkor lehetőség szerint az azt használó adattípus moduljában rejtünk el.

- **A reprezentáció elrejtése**

Az adatrejtés támogatására a nyelvek gyakran az összetett típusok komponenseinek láthatóságát szintekre bontják, ezek meghatározzák, hogy pontosan kik férhetnek hozzá az adott komponenshez.
Leggyakrabban három szintű:
*nyilvános* (public) – az adott komponens mindenki számára látható
*védett* (protected) – az adott komponens csak a leszármazottak számára látható
*privát* (private) – a reprezentáció teljesen rejtett része, csak a műveletek implementációjában használható komponensek

- Vannak olyan nyelvek ahol egyáltalán nincs ilyen jellegű szabályozás.
- Egyes nyelvekben (például az Eiffelben) a láthatóság még ennél is finomabban szabályozható, a típus megvalósításakor rendelkezhetünk arról, hogy mely osztály (és leszármazottai) férhessenek hozzá az adott komponenshez.

```
Class A
 feature {B}                          <-- B látja
   X: INTEGER;
 feature{ANY}                         <-- public
  make( p_name : STRING )  is
  require
      p_name /= ""
  do
      name := p_name ….
 feature{NONE}                        <-- private
   Y: ARRAY[INTEGER];
end
```

**CLU:** (public)

```
complex = cluster is newcomplex, re, im, add, addreal, sub, subreal,
                div, divreal, mul, mulreal, sqrt, sqrtreal, abs, phi, minus
  rep = struct[ r : real, i : real ]  % description of representation
  % implementation of the operations
  newcomplex = proc (nr: real, ni: real)returns ( cvt )
    return ( rep$_(r: nr, i: ni) )
  end newcomplex
  re = proc ( c : cvt ) returns ( real )  return ( rep$get_r( c ) )
  end re
  im = proc ( c : cvt ) returns ( real )  return ( rep$get_i( c ) )
  end im
  add = proc ( cl : cvt, c2 : cvt ) returns ( cvt )
     return (rep$_{re : rep$get_r(cl) + rep$get_r(c2),
                   im : rep$get_i(cl) + rep$get_i(c2)} )
  end add
  …
end complex
```

(private)

## Specifikáció és implementáció szétválasztása

Azon nyelvekben, melyek támogatják az “egy modul egy típus” elvet, gyakran lehetőség van a típusspecifikációnak az implementációtól külön, önálló fordítási egységben történő leírására. Ez segít abban, hogy az egyes modulok egymástól függetlenül elkészíthetőek legyenek, továbbá lehetővé teszi a reprezentáció és implementáció észrevétlen megváltoztatását.

A reprezentáció elrejtése sajnos *teljes egészében nem lehetséges*.

Bár a használó modul számára nem szükséges – sőt nem is tanácsos – az ismerete, de a fordító programnak tisztában kell lennie azzal, hogy mennyi memóriát kell egy, az adott típushoz tartozó objektum számára lefoglalni. Éppen ezért, ha a specifikációt és az implementációt két fordítási egységre bontjuk szét, akkor gondoskodnunk kell arról, hogy csak a specifikációs rész birtokában a fordító program képes legyen a szükséges memóriafoglalást meghatározni.

A probléma egyik lehetséges megoldása, hogy az absztrakt adattípus mindig egy **mutatóra** képződik le, amely aztán a ténylegesen reprezentáló bájtsorozatra mutat.

Ilyenkor a használat helyén a helyfoglalás pontosan meghatározható anélkül, hogy a tényleges reprezentációról bármit is elárulnánk, ám a használatot alaposan megnehezíti a bevezetett indirekció és a dinamikus memóriakezelés szükségessége. Ezt a megoldást támogatja például a Modula-2.

**MODULA-2:**

```
DEFINITION MODULE Complex_Numbers;
    TYPE Complex;
    TYPE Angle = REAL;
    PROCEDURE NewComplex(R,I: REAL):Complex;
    PROCEDURE DeleteComplex(VAR Z: Complex);
    PROCEDURE Add(Z1, Z2: Complex);
    PROCEDURE AddReal(Z: Complex; X: Float);
    PROCEDURE Mul(Z1, Z2: Complex);
    PROCEDURE MulReal(Z: Complex; X: Float);
    PROCEDURE Re(Z: Complex):REAL;
    PROCEDURE Im(Z: Complex):REAL;
    PROCEDURE Phi(Z: Complex):Angle;
    PROCEDURE Abs(Z: Complex):REAL;
     …..
END Complex_Numbers.
```

```
IMPLEMENTATION MODULE Complex_Numbers;
    CONST GUARD = 12345;
    TYPE Complex = POINTER TO ComplexStr;
    TYPE ComplexStr = RECORD
                        R    : REAL;
                        I    : REAL;
                        G    : CARDINAL;
                      END;
    PROCEDURE NewComplex(R,I: REAL):Complex
    VAR Z:Complex;
    BEGIN
      NEW(Z);
      Z^.R:=R; Z^.I:=I; Z^.G:=GUARD;
      RETURN Z;
    END;
    PROCEDURE Add(Z1, Z2: Complex);
    BEGIN
      IF (Z1#NIL) AND (Z2#NIL) AND
        (Z1^.G=GUARD) AND (Z2^.G=GUARD) THEN
        Z1^.R:=Z1^.R+Z2^.R; Z1^.I:=Z1^.I+Z2^.I;
      END;
    END Add;

  ...
  BEGIN
  END Complex_Numbers;
```

Az **Ada** egy teljesen más megközelítést választott. Itt a specifikáció és a reprezentáció került egy fordítási egységbe, így persze a reprezentáció *fizikailag* nincs elrejtve, de a nyelv, szintaktikus eszközeivel, gondoskodik arról, hogy ezen információkat a típust felhasználó másik programegység írója ne tudja kihasználni.

```
with Ada.Numerics; use Ada.Numerics;
Package Complex_Numbers is
 Type Complex is private;
   Type Angle is new Float range 0.0 .. 2.0*PI;
   Function NewComplex(R: Float; I: Float:=0.0) return
                                              Complex;
   Function "+"(Z1, Z2: Complex) return Complex;
   Function "+"(Z: Complex; X: Float) return Complex;
   Function "+"(X: Float; Z:Complex) return Complex;
   Function "*"(Z1, Z2: Complex) return Complex;
   Function "*"(Z: Complex; X: Float) return Complex;
   Function "*"(X: Float; Z:Complex) return Complex;
   Function "-"(Z1, Z2: Complex) return Complex;
   Function "-"(Z: Complex; X: Float) return Complex;
   Function "-"(X: Float; Z:Complex) return Complex;
   Function "/"(Z1, Z2: Complex) return Complex;
   Function "/"(Z: Complex; X: Float) return Complex;
   Function "/"(X: Float; Z:Complex) return Complex;
   Function Sqrt(X: Float) Return Complex;
   Function Sqrt(Z: Complex) Return Complex;
   Function Re(Z: Complex) return Float;
   Function Im(Z: Complex) return Float;
   Function Phi(Z: Complex) return Angle;
   Function Absval(Z: Complex) return Float;
```

-- a reprezentáció:

```
Private
   Type Complex is record
     R : Float := 0.0;
     I : Float := 0.0;
   End record;
End Complex_Numbers;
```

```
Package body Complex_Numbers is
     Function NewComplex(R: Float; I: Float:=0.0) return
   Complex is
     Begin
       Return Complex‘(R,I);
     End New;

     Function “+”(Z1, Z2: Complex) return Complex is
     Begin
       Return NewComplex(Z1.R+Z2.R, Z1.I+Z2.I);
     End “+”;

     Function “+”(Z: Complex; X: Float) return Complex is
     Begin
       Return Z1 + NewComplex(X);
     End “+”;

     Function “+”(X: Float; Z:Complex) return Complex is
…….
end Complex_Numbers;
```

Más nyelvekben (például Eiffel) a fizikai szétválasztás nem lehetséges – így persze a helyfoglalási probléma sem jelentkezik –, ám a fejlesztő eszköz támogatja a kód többszintű "nézetét", így a felhasználás szempontjából lényegtelen részletek eltakarhatóak.

A Java nyelvből már ez a lehetőség is hiányzik, itt a specifikáció és implementáció összemosódik. Egyetlen segítséget a javadoc program jelent, amely a megfelelően dokumentált programkódból elő tudja állítani a specifikáció szöveges leírását.

A C/C++ nyelvekben a specifikációt fizikailag be kell másolni minden olyan fordítási egységbe, amely az adott típust használni akarja, így persze itt sem beszélhetünk igazi szétválasztásról.

A bemásolás megkönnyítésére a specifikáció egy külön, speciális forrásfájlban (header fájl) leírható és az előfordító segítségével azt beemelhetjük a megfelelő fordítási egységekbe. Ha egy típus specifikációját többször is beírjuk egy fordítási egységbe, az hibát jelent, így ennek kivédéséről a header fájl megírásakor a programozónak gondoskodnia kell.

```
typedef double Angle;
class Complex {
   public:
    Complex(double r=0, double i=0){ R = r; I = i;}
    Complex operator =(Complex z){
                        R = z.R; I = z.I; return *this; }
    Complex operator +(Complex z) {
                        return  Complex(R+z.R,I+z.I);}
    Complex operator +(double x) { return Complex(R+x,I);}
    Complex operator *(Complex z);
    Complex operator *(double x);
    double Re();
    double Im();
    double Abs();
    Angle  Phi(); …
    private:
      double R;
      double I;
   }
```

……

```
Complex operator + (double x, Complex z)
   {
     return z+x
   }
```

**Vagy: friend**

```
…..
    #include "complex.h"

    Complex Exp(Complex z, double eps = 0.0001)
    {
      Complex zi = Complex(1.0);
      Complex sum;
      double  i  = 1.0;

      while(zi.Abs() >= eps )
      {
        sum = sum+zi;
        i  += 1.0;
        zi  = (zi*z)/i;
      }
      return sum;
    }
```

## Modulfüggőség kezelése

Egy program legmagasabb szintű építőkövei a modulok. A program működése ezen modulok interakciója. Minden modul igényel szolgáltatásokat és segítségükkel más szolgáltatásokat valósít meg, így a modulok között egyfajta függőségi reláció alakul ki, s egy modul megváltozása esetén szükségessé válhat a tőle függő modulok újrafordítása.

Némelyik programozási nyelv (például a C vagy C++) teljes egészében a programozóra bízza ezen függőségek kezelését, minden fordítási egységet önállóan kezel. Éppen ezért vannak speciális eszközök (pl. make), amelyek kizárólagos feladata ennek a feladatnak a megkönnyítése. Ezek a megoldások nem tökéletesek, előfordul, hogy túl sokszor fordítanak újra valamit, vagy éppen nem veszik észre az újrafordítás szükségességét stb.

Más nyelvekben a függőségek kezelése a fordító program feladata. Az Ada nyelvben például a with utasítással kell megadnunk, hogy milyen más fordítási egységektől függ a szóban forgó modul. Ez garantálja azt, hogy a modul fordítása előtt mindazon modulok specifikációs része lefordításra kerül (ha az szükséges), amelyektől az adott modul függ.

- **Konzisztens használat**
  - Ne tegyen különbséget a beépített és a programozó által definiált típusok között a használat szempontjából! Ugyanúgy lehessen összetett típusokat (tömböket, rekordokat stb.) definiálni a segítségükkel, ugyanúgy lehessen változókat definiálni a saját típusokkal stb.
    ⇒ az új típust be lehessen illeszteni a nyelv logikájába. Ha például az adott nyelvben az a konvenció, hogy egy típust a Read művelet olvas be, akkor fontos, hogy a saját típusokhoz is definiálhasson a programozó egy Read nevű műveletet, azaz legyen lehetőség a Read azonosító *túlterhelésére* vagy *átlapolására*.
  - Egy azonosító *túlterhelése* vagy *átlapolása* (overloading) azt jelenti, hogy a programszöveg egy adott pontján az azonosítónak több definíciója is érvényben van.
  - Speciálisan az *operátor-túlterhelésnek* nagy jelentősége van abban, hogy a felhasználói típusokat természetes használatuknak megfelelően használhassuk.

## Kérdések:

- Saját adattípus önálló fordítási egységként megvalósítható?
- Specifikáció és implementáció különválasztható?
- Reprezentáció elrejtése megvalósítható?
- Milyen láthatósági szintek vannak?
- Van-e azonosító túlterhelés/ operátor túlterhelés/ free operátor?
- Beépített típusokkal megegyező módon használható?