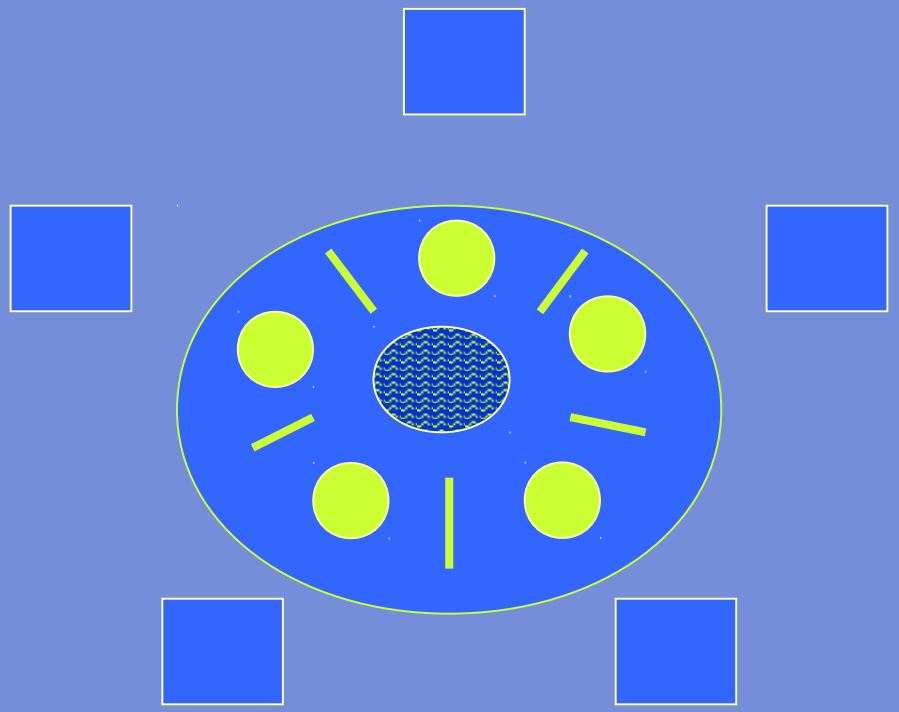
12. előadás

Szinkronizációs feladatok (2). Védett egységek. Álvéletlenszámok. OOP az Adában.

Az evő filozófusok problémája

- Dijkstra: dining philosophers
- Erőforráshasználat modellezése
  - Erőforrások: villák
  - Folyamatok: filozófusok
- Holtpont kialakulása
- Általánosítás: ivó filozófusok



Holtpont

- Deadlock
- Ha a folyamatok egymásra várnak
- Tranzakciók
- Elkerülés
  - Például a szimmetria megtörésével
    - Erőforrások lekötése rendezés szerint
    - **Véletlenszerű időkorlátok**
- Felismerés és megszüntetés (abort)

A villák (erőforrások) modellezése

```
task type Villa is
        entry Felvesz;
        entry Letesz;
end Villa;
task body Villa is
begin
        loop
                accept Felvesz;    accept Letesz;
        end loop;
end Villa;
```

A filozófusok (folyamatok)

```
Villák: array (1..5) of Villa;
task type Filozófus ( Bal, Jobb: Positive );

Platon: Filozófus(1,2);
Arisztotelész: Filozófus(2,3);
Descartes: Filozófus(3,4);
Kant: Filozófus(4,5);
Hegel: Filozófus(5,1);
```

Véletlenszerű időkorlát (1)

```
task body Filozófus is
        procedure Éhes_Vagyok is … end Éhes_Vagyok;
begin
        loop
                -- gondolkodom
                Éhes_Vagyok;
                -- eszem
                Villák(Bal).Letesz;
                Villák(Jobb).Letesz;
        end loop;
end Filozófus;
```

Véletlenszerű időkorlát (2)

```
procedure Éhes_Vagyok is
    Ehetek: Boolean := False;
begin
    while not Ehetek loop
        Villák(Bal).Felvesz;
        select
            Villák(Jobb).Felvesz;    Ehetek := True;
        or  delay Véletlen_Idő;      Villák(Bal).Letesz;
        end select;
    end loop;
end Éhes_Vagyok;
```

„Véletlenszámok”

- Álvéletlenszámok (pseudo-random numbers)
- Valamilyen (determinisztikus) algoritmussal
- Float vagy diszkrét típusú
  - Ada.Numerics.Float_Random (csomag)
  - Ada.Numerics.Discrete_Random (sablon csomag)
- Generator, Reset, Random stb.
- Szimulációkhoz
- Taszkok esetén: szinkronizáltan

Float_Random

```
with Ada.Numerics.Float_Random;
use Ada.Numerics.Float_Random;
…
        G: Generator;
…
        Reset(G);                       Reset(G,100);
…
        F: Float := Random(G);
…
```

Discrete_Random (egész)

```
with Ada.Numerics.Discrete_Random;
…
   type Intervallum is new Integer range 1..100;
   package Intervallum_Random is
         new Ada.Numerics.Discrete_Random(Intervallum);
   use Intervallum_Random;
   G: Generator;
…
   Reset(G);                    Reset(G,100);
…
   F: Intervallum := Random(G);
…
```

Discrete_Random (felsorolási)

```
with Ada.Numerics.Discrete_Random;
…
   type Alapszín is (Piros, Sárga, Kék);
   package Alapszín_Random is
         new Ada.Numerics.Discrete_Random(Alapszín);
   use Alapszín_Random;
   G: Generator;
…
   Reset(G);                        Reset(G,100);
…
   F: Alapszín := Random(G);
…
```

Monitorral védetten

```
task body Véletlen is
        G: Ada.Numerics.Float_Random.Generator;
begin
        Reset(G);
        loop
                select
                        accept Random ( F: out Float ) do
                                F := Random(G);
                        end Random;
                or      terminate;
                end select;
        end loop;
end Véletlen;
```

```
task Véletlen is
        entry Random ( F: out Float );
end Véletlen;
```

A véletlenszerű várakozáshoz

```
Villák: array (1..5) of Villa;

task Véletlen is entry Random( F: out Float ); end;  …
function Véletlen_Idő return Duration is
    F: Float;
begin
    Véletlen.Random(F);    return Duration(F);
end Véletlen_Idő;

task type Filozófus ( Bal, Jobb: Positive );
…
```

Író-olvasó feladat

- Van egy több folyamat által használt erőforrás
- Lehet változtatni („írni”) és lekérdezni („olvasni”)
- Az olvasások mehetnek egyidőben
- Az írás kizárólagos
- A monitornál megengedőbb

Megvalósítás Adában

- Taszkok segítségével
  - Bonyolult
  - Kell pár select bele
  - Igazságosság, pártatlanság (fairness)
- Védett egységgel

Kiéheztetés

- Livelock, starvation
- Nincs holtpont (deadlock)
- Fut a rendszer
- De: egy folyamat mindig rosszul jár
- Nem képes elvégezni a feladatát
- Példa: ügyetlen „evő filozófusok” megvalósításnál
  - Szimmetria megtörése a filozófusok beszámozásával
  - A kisebb sorszámú előnyt élvez a nagyobb sorszámúval szemben
- Példa: ügyetlen „író-olvasó” megvalósításnál
  - Az olvasók könnyen kiéheztethetik az írókat

Író-olvasó: Ada taszk segítségével (1)

```
task Scheduler is
        entry Start_Reading;
        entry Start_Writing;
        entry Stop_Reading;
        entry Stop_Writing;
end Scheduler;
```

- Szemafor-szerű

Író-olvasó: Ada taszk segítségével (2)

```
task body Reader_Writer is
begin
    loop
        Scheduler.Start_Reading;
        -- itt olvashatom az erőforrást
        Scheduler.Stop_Reading;
        …
        Scheduler.Start_Writing;
        -- itt írhatom az erőforrást
        Scheduler.Stop_Writing;
    end loop;
end Reader_Writer ;
```

Író-olvasó: Ada taszk segítségével (3)

```
task body Scheduler is
    Nr_Of_Readers: Natural := 0;
    Writing: Boolean := False;
begin
    loop
        select   … accept Start_Reading  …
        or       … accept Start_Writing  …
        …
        or terminate;
        end select;
    end loop;
end Scheduler;
```

Író-olvasó: Ada taszk segítségével (4)

```
select
    when not Writing =>
                    accept Start_Reading;
                    Nr_Of_Readers := Nr_Of_Readers + 1;
or when (not Writing) and then (Nr_Of_Readers = 0) =>
                    accept Start_Writing;   Writing := True;
or accept Stop_Reading;
    Nr_Of_Readers := Nr_Of_Readers - 1;
or accept Stop_Writing;   Writing := False;
or terminate;
end select;
```

az írók éhezhetnek

Író-olvasó: Ada taszk segítségével (5)

```
select
    when (not Writing) and then (Start_Writing'Count = 0) =>
                    accept Start_Reading;
                    Nr_Of_Readers := Nr_Of_Readers + 1;
or when (not Writing) and then (Nr_Of_Readers = 0) =>
                    accept Start_Writing;   Writing := True;
or accept Stop_Reading;
    Nr_Of_Readers := Nr_Of_Readers - 1;
or accept Stop_Writing;   Writing := False;
or terminate;
end select;
```

az olvasók éhezhetnek

Író-olvasó: Ada taszk segítségével (6)

- Egy ügyesebb stratégia:
  - Egy sorban tároljuk a bejövő olvasási és írási kérelmeket
  - A sor elején álló olvasókat (az első íróig) engedjük egyszerre olvasni
  - Ha a sor elején író áll, akkor engedjük írni
- Ebben nincs kiéheztetés
- Bizonyos olvasó folyamatokat nem engedek olvasni, amikor pedig épp „olvasási szakasz” van

Író-olvasó védett egységgel

- Sokkal egyszerűbb
- Nincs szükség Scheduler taszkra
- Nincs szükség az írókban és olvasókban a kritikus szakasz kijelölésére
- Az erőforrás köré szinkronizációs burok
- Védett egység
- A monitor továbbfejlesztése író-olvasóhoz

A védett egység

```
protected Erőforrás is
        function Olvas return Adat;
        procedure Ír( A: in Adat );
private
        X: Adat;
end Erőforrás;
…
protected body Erőforrás is
        function Olvas return Adat is begin return X; end;
        procedure Ír( A: in Adat ) is begin X := A; end;
end Erőforrás;
```

Szinkronizáció a védett egységre

- A függvények egyidőben
- Az eljárásokra kölcsönös kizárás
  - egy eljárással egyidőben nem lehet:
    - sem másik eljárás
    - sem függvény
- Feltételezés:
  a függvények mellékhatásmentesek

Védett egység és monitor

```
task Monitor is
    entry M1 ( ... );
    entry M2 ( ... );
end Monitor;

task body Monitor is
begin
    loop
        select
            accept M1 ( ... ) do … end;
        or  accept M2 ( ... ) do … end;
        or terminate;
        end select;
    end loop;
end Monitor;
```

```
protected Monitor is
    procedure M1 ( ... );
    procedure M2 ( ... );
end Monitor;

protected body Monitor is
    procedure M1 ( ... ) is … end;
    procedure M2 ( ... ) is … end;
end Monitor;
```

```
…
    Monitor.M1(…);
    Monitor.M2(…);
…
```

Belépési pontok

- Az eljárások és függvények mellett
- Kölcsönös kizárás teljesül rá
- Feltétel szabható rá
- Várakozási sor

```
entry E (…) when Feltétel is
    …
begin
    …
end E;
```

nem függhet a paraméterektől

Termelő-fogyasztó (korlátlan buffer)

```
protected Osztott_Sor is
    procedure Betesz ( A: in Adat );
    entry Kivesz ( A: out Adat );
private
    S: Sor;
end Osztott_Sor;
```

```
protected Osztott_Sor is
    procedure Betesz ( A: in Adat )
    is
    begin
        Betesz(S,A);
    end Betesz;

    entry Kivesz ( A: out Adat )
    when not Üres(S) is
    begin
        Kivesz(S,A);
    end Kivesz;
end Osztott_Sor;
```

Védett objektum és védett típus

- Mint a taszk és a taszk típus
- A típus
  - korlátozott
  - diszkrimináns(oka)t tartalmazhat (diszkrét vagy mutató)
  - cím szerint átadandó

```
protected type Osztott_Sor( Kapacitás: Positive ) is
    entry Betesz ( A: in Adat );
    entry Kivesz ( A: out Adat );
private
    S: Sor(Kapacitás);
end Osztott_Sor;
```

Védett egységek

- Programegységek
- Specifikáció és törzs szétválik (csomag, sablon, taszk)
- Hasonlóság a taszkokkal: nem lehet könyvtári egység, csak beágyazva
- Hasonlóság a csomagokkal: private rész
- Megkötések: nem lehet benne például típus vagy csomag definíciója

„Szervernek” megírt taszkok helyett

```
task (type) T is
    entry P;
    entry E;
end T;
task body T is
    X: Adat;
begin
    loop
        select
            accept P …
        or  when Feltétel => accept E …
        or  terminate;
        end select;
    end loop;
end T;
```

```
protected (type) T is
    procedure P;
    entry E;
private
    X: Adat;
end T;

protected body T is
    procedure P is …
    entry E when Feltétel is …
end T;
```

Taszkok versus védett egységek

- Monitor taszk helyett védett egység: tömörebb, kényelmesebb
- Hatékonyabb
  - nincs külön végrehajtási szál
  - a hívott kód végrehajtása a hívó taszkban
  - a védett egység passzív, csak szinkronizál
  - kevesebb kontextusváltás kell
- Író-olvasó is triviális

Működési elv: író-olvasó

- Függvény mehet, ha
  - nem fut eljárás vagy entry törzse
- Eljárás mehet, ha
  - nem fut függvény, eljárás vagy entry törzse
- Entry törzse mehet, ha
  - nem fut függvény, eljárás vagy entry törzse
  - és az őrfeltétel teljesül

Várakozási sorok

- Minden belépési ponthoz
- Ha a védett egységbe belépéskor az őrfeltétel hamis, ide kerül a folyamat
- Ha véget ér egy eljárás vagy entry törzs, az őrfeltételek újra kiértékelődnek

Kétszintű várakoztatás

- A folyamatok várni kényszerülhetnek arra,
  - hogy a védett egységbe bejussanak (mutex)
  - egy entry törzsébe bejussanak (őrfeltétel)
- Egy entry várakozási sorában álló folyamat előnyben van a még be sem jutottakkal szemben
- Példa: üzenetszórásos szignál

Üzenetszórásos szignál

- Broadcast signal
- Folyamatok várnak egy eseményre
- Blokkolódnak, amíg az esemény be nem következik
- Az esemény bekövetkeztekor szignál
- Ada megvalósítás: védett egység + requeue

A requeue utasítás

- Egy entry törzsében használható
- A hívást átirányítja
  - egy másik belépési ponthoz
  - akár másik védett egységbe vagy taszkba
  - akár visszairányítja önmagára
- A másik entry:
  - vagy paraméter nélküli
  - vagy ugyanazokkal a paraméterekkel

Üzenetszórásos szignál Adában (1)

```
protected Esemény is
    entry Vár;      -- itt állnak sorban, akik az üzenetet
    szeretnék
    entry Jelez;    -- jelzés, hogy bekövetkezett
private
    entry Alaphelyzet;      -- lokális belépési pont
    Megtörtént: Boolean := False;
end Esemény;
```

Üzenetszórásos szignál Adában (2)

```
protected body Esemény is
    entry Vár when Megtörtént is begin null; end Vár;

    entry Jelez when True is
    begin
        Megtörtént := True;
        requeue Alaphelyzet;
    end Jelez;

    entry Alaphelyzet when Vár'Count = 0 is
    begin
        Megtörtént := False;
    end Alaphelyzet;
end Esemény;
```

Belépésipont-családok

- Belépési pontokból álló tömb
- Taszkban és védett egységben is lehet

```
type Prioritás is (Magas, Közepes, Alacsony);
task Kijelző is
    entry Üzenet(Prioritás) ( Szöveg: in String );
end Kijelző;

accept Üzenet(Magas) ( Szöveg: in String ) do ... end;

Kijelző.Üzenet(Magas)(”Nemsokára ZH!”);
```

Védett egységben entry-család (1)

```
type Prioritás is (Alacsony, Közepes, Magas);
protected Kijelző is
    entry Üzenet (Prioritás) (Szöveg: in String );
end Kijelző;
```

- Hívás:

```
Kijelző.Üzenet(Magas)("Nemsokára ZH!");
```

Védett egységben entry-család (2)

```
protected body Kijelző is

   function Mehet ( Aktuális: Prioritás ) return Boolean is…

   entry Üzenet (for Aktuális in Prioritás) (Szöveg: in String)
   when Mehet(Aktuális) is
   begin
         Put_Line(Prioritás'Image(Aktuális) & Ascii.HT & Szöveg);
   end Üzenet;

end Kijelző;
```

Védett egységben entry-család (3)

```
function Mehet ( Aktuális: Prioritás ) return Boolean is
    Fontosabb: Prioritás := Prioritás'Last;
begin
    while Fontosabb > Aktuális loop
        if Üzenet(Fontosabb)'Count > 0 then
            return False;
        end if;
        Fontosabb := Prioritás'Pred(Fontosabb);
    end loop;
    return True;
end Mehet;
```

Egymásba ágyazott randevúk

- Egy taszk törzsében bárhol elhelyezhető accept utasítás
  - de csak a törzsben, alprogramjában már nem
- Akár még egy randevún belül is
  - *„accept E”* –ben nem lehet másik *„accept E”*
- Sőt, egy hívás is lehet randevún belül
- Nagyon veszélyes, több taszk is várni kezd
  - Holtpont is könnyen kialakulhat
- Megfontoltan csináljunk csak ilyet
  - Pl. kössük időkorláthoz a belső randevút
- Védett egységben egyáltalán ne csináljunk ilyet

OOP az Adában

- Adatabsztrakció, egységbe zárás
- Konstruktor-destruktor mechanizmus
- Öröklődés (+kiterjesztés)
- Műveletek felüldefiniálása, dinamikus kötés
- Altípusos polimorfizmus

- Nem szokványos módon

Adatabsztrakció, egységbe zárás

- Csomagok segítségével
- Átlátszatlan típus definiálható
- Külön fordítható műveletmegvalósítások
- Paraméteres adattípus: sablon csomagban

- Egység biztonságos kinyitása: gyerekcsomag

Konstruktor-destruktor

- A Controlled, illetve Limited_Controlled típusokkal
- Initialize, Adjust és Finalize eljárások
- Tipikusan láncolt adatszerkezetek megvalósításához
- Az Initialize nem létrehoz paraméterek alapján, hanem „rendbetesz”

Öröklődés, felüldefiniálás

- Típusszármaztatással
- A típusértékhalmaz és a primitív műveletek lemásolódnak
- Műveletek felüldefiniálhatók
- Újabb műveletekkel kiterjeszthető
- Publikus és privát öröklődés is lehetséges

- Adattagokkal való kiterjesztés?

Jelölt (tagged) rekordok

- Származtatásnál újabb mezőkkel bővíthető
- Cím szerint átadandó típus

```
type Alakzat is tagged record
                        Színe: Szín;
                    end record;

type Pont is new Alakzat with record
                            X, Y: Float;
                        end record;
```

Típusszármaztatás és konverzió

- A típusszármaztatás új típust hoz létre
- A régi és az új között konvertálhatunk
- Csak explicit konverzióval

```
type Int is new Integer;
I: Integer := 3;
J: Int := Int(I);
K: Integer := Integer(J);
```

Konverzió jelölt típusokra

```
type Alakzat is tagged record …
type Pont is new Alakzat with record …
P: Pont := (Piros, 3.14, 2.73);
A: Alakzat := Alakzat(P);      -- csonkol

P := (A with X => 2.73, Y => 3.14);
```

Altípusok

- A subtype mechanizmussal altípusok definiálhatók
- A típusértékhalmaz csökkenthető
- Csak futási idejű ellenőrzésekhez
- A fordító számára ugyanaz a típus

- Öröklődés és altípusképzés?

Altípusos polimorfizmus

- Osztályszintű típusok: 'Class attribútum
- T'Class: a T jelölt típus és leszármazottai
- Únióza azok típusértékhalmazát
- Implicit konverzió:
  Pont konvertálódik Alakzat'Class-ra
- Nem teljesen definiált típus
  X: Alakzat'Class := Pont'(Piros, 3.14, 2.73);
- Alakzat'Class altípusa Pont'Class

Heterogén szerkezetek

```
X: array (1..3) of Alakzat :=
        ( (Színe=>Piros), (Színe=>Fehér), (Színe=>Zöld) );
```

csak Alakzatok

akár Pontok is

```
type Alakzatok_Access is access Alakzat'Class;
Y: array (1..3) of Alakzatok_Access :=
                        (   new Pont'(Piros, 3.14,  2.73),
                            new Alakzat'(Színe=>Fehér),
                            new Alakzat'(Színe=>Zöld)     );
```

Statikus és dinamikus típus

X: Alakzat'Class
:= Bekér;

- statikus: Alakzat'Class
- dinamikus: Alakzat / Pont
- nem változhat a dinamikus típus

```
function Bekér return Alakzat'Class is
    P: Pont;
begin
    Get( P.Színe );
    if End_Of_Line then
            return Alakzat(P);
    else
            Get( P.X );  Get( P.Y );
            return P;
    end if;
end Bekér;
```

Dinamikus kötés

- Felüldefiniálás esetén
- A dinamikus típus szerinti kód hajtódik végre
- A kötés végrehajtási időben
- Szemben a statikus kötéssel (fordító)
- Kisebb hatékonyság, nagyobb rugalmasság
- Lehetőségek:
  - Java: mindig dinamikus
  - C++: a művelet definiálásakor jelzem (virtual)
  - Ada: a híváskor jelzem

Primitív művelet felüldefiniálása (1)

```
package Alakzatok is
    type Alakzat is tagged record Színe: Szín; end record;
    function Image ( A: Alakzat ) return String;

    type Pont is new Alakzat with record
                                    X, Y: Float;
                                end record;
    function Image ( P: Pont ) return String;
end Alakzatok;
```

Primitív művelet felüldefiniálása (2)

```
package body Alakzatok is

    function Image ( A: Alakzat ) return String is
    begin   return Szín'Image(A.Színe);  end Image;

    function Image ( P: Pont ) return String is
    begin
          return  Image(Alakzat(P)) & ' ' &
                        Float'Image(P.X) & ' ' & Float'Image(P.Y);
    end Image;

end Alakzatok;
```

Dinamikus kötés híváskor

```
A: Alakzat := (Színe => Piros);
P: Pont := (Piros, 3.14, 2.73);
X: Alakzat'Class := Bekér;
Y: Alakzatok_Access := new Pont'(Fehér, 2.73, 3.14);
…
Put_Line( Image(A) );                    -- statikus kötés
Put_Line( Image(P) );                    -- statikus kötés
Put_Line( Image(X) );                    -- dinamikus
   kötés
Put_Line( Image(Y.all) );                -- dinamikus kötés
```

Ugyanaz a művelet stat./din. kötéssel

```
procedure Put_Statikus ( A: in Alakzat ) is
begin
        Put( Image(A) );
end Put_Statikus;

procedure Put_Dinamikus ( A: in Alakzat ) is
begin
        Put( Image(Alakzat'Class(A)) );
end Put_Dinamikus;
```

Újrakiválasztás (1)

```
package Alakzatok is
    type Alakzat is tagged record Színe: Szín; end record;
    function Image ( A: Alakzat ) return String;
    procedure Put ( A: in Alakzat );

    type Pont is new Alakzat with record
                                    X, Y: Float;
                                end record;
    function Image ( P: Pont ) return String;
end Alakzatok;
```

Újrakiválasztás (2)

```
package body Alakzatok is

    function Image ( A: Alakzat ) return String is
    begin   return Szín'Image(A.Színe);  end Image;

    procedure Put ( A: in Alakzat ) is
    begin   Put( Image(Alakzat'Class(A)) );  end Put;

    function Image ( P: Pont ) return String is
    begin
          return  Image(Alakzat(P)) & ' ' &
                        Float'Image(P.X) & ' ' & Float'Image(P.Y);
    end Image;

end Alakzatok;
```

Egyenlőség osztályszintű típusokra

```
X, Y: Alakzat'Class := Bekér;
…
if   X.all = Y.all   then … end if;
```

- Ha különbözik a dinamikus típus, akkor False (nem Constraint_Error)

Dinamikus típusellenőrzés

- Egy értéket tartalmaz-e egy típus
- Mint a subtype-oknál

X: Alakzat'Class := Bekér;

X in Pont'Class -- típustartalmazás

X in Pont -- típusegyezés

Absztrakt típus

- Nem használható objektum létrehozására
- Absztrakt (megvalósítás nélküli) műveletet tartalmazhat
- Típusspecifikáció vagy részleges implementáció
- Absztrakció: a leszármazottak közös őse

```
type Alakzat is abstract tagged record Színe: Szín; end
    record;
function Image ( A: Alakzat ) return String;
procedure Kirajzol ( A: in Alakzat ) is abstract;
```

Többszörös öröklődés

- Különböző vélemények
  - nincs (Simula, Smalltalk, Ruby, VB, Beta, Oberon ...)
  - teljes támogatás (CLOS, C++, Eiffel …)
  - típusspecifikációra (Objective-C, Object Pascal, Java, C# ...)
- Egyáltalán nincs az Adában
- Helyette más lehetőségek (mixin)
  - egyszeres öröklődés + sablon
  - access diszkrimináns