Programozási nyelvek és módszerek 2010.

1. tétel *Szoftverek minőségi szempontjai, …*

Szoftverek minőségi szempontjai:

- helyesség: ha pontosan megoldja a feladatot és megfelel a specifikációnak.
- megbízhatóság: ha helyes, és abnormális helyzetekben is „ésszerű” működést produkál.
- karbantarthatóság: annak mérőszáma, hogy mennyire könnyen lehet adaptálni a programot a specifikáció változtatásához.
- újrafelhasználhatóság: egészben vagy részben felhasználható más alkalmazásokban.
- kompatibilitás: mennyire könnyű a különböző szoftvereket egymással kombinálni.
- hordozhatóság: mennyire könnyű a programot más környezethez alakítani.
- hatékonyság: mennyire fut gyorsan, mennyi memóriát használ.
- barátságosság: a bevitel logikus és egyszerű, a kimenet áttekinthető.
- tesztelhetőség

Moduláris tervezés: a programokat minél inkább független, jól definiált programegységekként tervezzük.

Moduláris dekompozíció: egy feladat több, egyszerűbb részfeladatra bontása, megoldásuk egymástól függetlenül is elvégezhető.

- a feladat bonyolultsága csökken
- az alrendszerek tovább dekomponálhatók
- fával ábrázolható

Moduláris kompozíció: egymással szabadon kombinálható szoftverelemek létrehozását támogatja.

Moduláris érthetőség: a modulok önállóan is egy-egy értelmes egységet alkotnak.

Moduláris folytonosság: a specifikáció kis változtatása esetén a programban is csak kis változtatásra van szükség.

Moduláris védelem: a program egészének védelme az abnormális helyzetek hatásaitól.
(a hiba hatása néhány modulra korlátozódjon)

A modularitás alapelvei:

1) **A modulokat nyelvi egységek támogassák**
   A modulok illeszkedjenek a használt nyelv szintaktikai egységeihez.
2) **Kevés kapcsolat a modulok között**
3) **A kapcsolatok gyengék** (minél kevesebb információt cseréljenek)
4) **Explicit interfész**
   Ha két modul kommunikál, annak legalább egyikük specifikációjából ki kell derülnie.
5) **Információ elrejtése** (a kifejezetten nyilvánosak kivételével minden rejtett)
6) **Nyitott és zárt modulok**
   nyitott: kiterjeszthető, így módosíthatóak eddigi szolgáltatásai
   zárt: jól definiált felületen érhető el, nem terjeszthető ki

Az újrafelhasználhatóság igényei:

1) A modulok többféle típusra is működjenek
2) Adatstruktúrák és algoritmusok változatossága
3) Egy típus – egy modul
4) Reprezentációfüggetlenség

2. tétel *Nyelvi elemek, alapfogalmak, lexikális elemek*

Def. Szintaxis: azon szabályok összessége, amelyek az adott nyelven írható összes formailag helyes programot definiálják, *pl. reguláris kifejezések.*

Def. Szemantika: az adott nyelv programjainak jelentését leíró szabályok összessége.

Interpreter: fordítás után azonnal végrehajt, fordítóprogram: csak fordít

forrásprogram ——*fordítás*——→ tárgyprogram

Def. Ábécé: tetszőleges szimbólumok véges, nem üres halmaza. Jele Σ.

Def. Szó: az ábécé elemeiből képzett $a_1a_2...a_k$ sorozat, ahol $k \geq 0$ és $\forall a_i \in \Sigma$.

Def. Σ^* az ábécé elemeiből képezhető összes szó halmaza.

Def. Σ^+ a nem üres szavak halmaza.

Def. Generatív nyelvtan: $G = (N, \Sigma, P, S)$, ahol:
N: nemterminálisok (grammatikai jelek) ábécéje (egyfajta szintaktikai változók)
Σ: terminálisok ábécéje, diszjunkt *N*-nel (tulajdonképpen maguk a betűk)
$S \in N$ kezdőszimbólum (pl. <mondat>, minden levezetés ezzel kezdődik)
P: átírási szabályok véges halmaza (pl. <mondat> → <alany> <állítmány>)

Def. Közvetlen levezetés: Legyen $\gamma, \delta \in (N \cup \Sigma)^*$.
$\gamma \Rightarrow_G \delta$ ha $\exists\ \alpha \rightarrow \beta \in P$ szabály
és $\exists\ \alpha', \beta' \in (N \cup \Sigma)^*$ szavak, hogy
$\gamma = \alpha'\alpha\beta'$ és $\delta = \alpha'\beta\beta'$
$\Rightarrow_G^n$ n lépéses levezetés
$\Rightarrow_G^+$ legalább 1 lépéses levezetés
$\Rightarrow_G^*$ levezethető (esetleg 0 lépésben is)

Def. G grammatika által generált nyelv: $L(G) = \{u \in \Sigma^* \mid S \Rightarrow_G^* u\}$
(azon szavak, amelyek a kezdőszimbólumból következnek)

Def. Ekvivalens nyelvtanok: Legyen $G = (N, \Sigma, P, S)$ és $G' = (N', \Sigma', P', S')$.
G ekvivalens G'-vel, ha $L(G) = L'(G')$.

Def. Chomsky nyelvosztályok:

0. *Kifejezés struktúrájú:* nincs korlátozás
1. *Környezetfüggő:* P-ben minden szabály „$(N \cup \Sigma)^* \rightarrow (N \cup \Sigma)^*$” alakú.
2. *Környezetfüggetlen:* P-ben minden szabály „$N \rightarrow (N \cup \Sigma)^*$” alakú.
3. *Reguláris:* P-ben minden szabály „$N \rightarrow N\,\Sigma^*$” vagy „$N \rightarrow \Sigma^*$” alakú.

Def. Levezetési fa: egy G grammatika szerinti levezetés olyan fával ábrázolható, melynek:

- gyökere a kezdőszimbólum,
- levelei balról jobbra a levezetett szó betűi,
- csomópontjai nemterminálisok,
- az elágazások az alkalmazott szabálynak felelnek meg.

Def. Programegység: egy részfeladatot megoldó (funkcionálisan összefüggő) programrész.

Def. Fordítási egység: a nyelv azon egysége, amely a program többi részétől elkülönülten lefordítható. Programegységek, lexikális elemek összessége.

Def. Lexikális elem: karaktersorozat határoló jelekkel elválasztva.
grafikus karakterek (a,b, 2, #, ...), *formátumvezérlő karakterek* (NL, CR, ...), *egyéb karakterek*

Def. Karakterkészlet: különböző karakterek egy halmaza. (név + megjelenés)

Def. Karakterkód: leképezés: karakterkészlet elemei ↔ természetes számok
Egyedi számkódot rendel a karakterkészlet minden karakteréhez.

Def. Karakterkódolás: algoritmus: karakterkód → oktetek sorozata

Numerikus literálok: 0..9 és A..F, néha _
Egészek: [-]{számjegy}+
Valósak: [-]{számjegy}+.{számjegy}+[kitevő]

Karakterek, karaktersorozatok:
Karakterek ('A'), „escape szekvenciák" ('\n')
Karakterláncok (stringek)

Megjegyzések: speciális jeltől a sor végéig / speciális kezdőjeltől speciális végejelig

3. tétel *Vezérlési szerkezetek, utasítások*

Értékadás: általánosan: [változó] [értékadásjel] [kifejezés]
Az értékadás kifejezés?

Kifejezés: operandusokból és operátorokból áll, pl. A + 5
kiértékeljük és megkapjuk az értékét

Szekvencia: több utasítás egymás után, egy sorban
a ; elválasztja, lezárja az utasításokat (C++, Java: a blokkutasítást nem)

Blokk: több utasítást foglal egyetlen utasításba.
Ada begin, end
C++ **Java** {}

Feltétel nélküli vezérlésátadás: *goto:* címkéhez ugrik (Javában nincs)

Elágazás: feltétel igaz vagy hamis értékétől függően valamilyen utasítás(csoport) végrehajtása
csellengő else: mindig a legbelső *if*-hez tartozik

az utasításcsoport blokkutasítás

többirányú elágazás: elsif

Ada *end if* kell a végén, *elsif* megengedett, nem kell blokkutasítás

C++ aritmetikai kifejezés kell (nem 0: true), blokk kell

Java logikai (boolean) kifejezés kell, blokk kell

Esetkiválasztásos elágazás: valamilyen kifejezés (ún. szelektor) értékétől függően osztályoz

Szelektor típusa? Minden lehetséges értéke kell? Mi történik, ha olyat vesz fel, amit nem soroltunk fel? „Rácsorog" a következő kiválasztási ágakra is? Diszjunktnak kell lenniük a kiválasztási értékeknek? Mi állhat a feltételben?

Ada

```
case expr is
      when choice1 => statements1;
      ...
      when others => statements;
end case;
```

others kötelező, ha nincs minden lefedve; nem csorog végig

C++ **Java** végigcsorog, *break* kell

Ciklusok: utasítások ismételt végrehajtása valamilyen feltételtől függően

Vannak nem ismert lépésszámú ciklusok? Van elöl/hátultesztelős ciklus? A ciklusfeltétel kötelezően logikai értékű? A ciklusmag kötelezően blokk? Beállítható a ciklusváltozó alsó és felső értéke, lépésszáma? Mi lehet a típusa? Változtatható-e a ciklusmagon belül? Mi a hatásköre, definiált-e az értéke kilépéskor? Van-e végtelen ciklus? Létezik-e break, continue*? Van-e ciklusváltozó-iterátor?*

Elöltesztelős ciklus:

Ada a ciklus utasítássorozat is lehet, a kifejezés logikai típusú

```
while kifejezés loop
      ciklusmag
end loop
```

C++ a kifejezés bármilyen típusú lehet, nem 0: true, 0: false

Java a kifejezés csak logikai típusú lehet

Hátultesztelős ciklus: a ciklusmag egyszer biztosan lefut

nincs szükség blokkutasításra: a ciklusmag vége meghatározott

Előre ismert lépésszámú ciklus: „for" ciklus

Minden végrehajtás után újra kiértékeli a lépésközt és a határt? A határt a ciklusmag előtt vagy után ellenőrzi?

Ada

```
for var in loop-range loop
      utasítások
end loop;
```

diszkrét típusú index, lokális konstans, az értékintervallumot belépés előtt számolja ki

reverse: fordított sorrend

Java lehet: `for ( int i : arrayOfInts )`

Végtelen ciklus:

Ada

```
loop
      utasítás
      exit when feltétel;
end loop;
```

Vezérlésátadó utasítások:

break: kiugrik a vezérlési szerkezetből
continue: ciklusfeltétel újrakiértékelése
return: visszatérés a hívóhoz
goto **Java** nincs!

4. tétel *Típusok 1.*

Adattípus: értékhalmaz + művelethalmaz.

Specifikáció: pl. egészen -∞-től ∞-ig, műveletek: +,-,*,/,stb.
Reprezentáció: 8/16/32/64 bit, ...
Implementáció: műveletek megvalósítása

Típusspecifikáció: „mit”

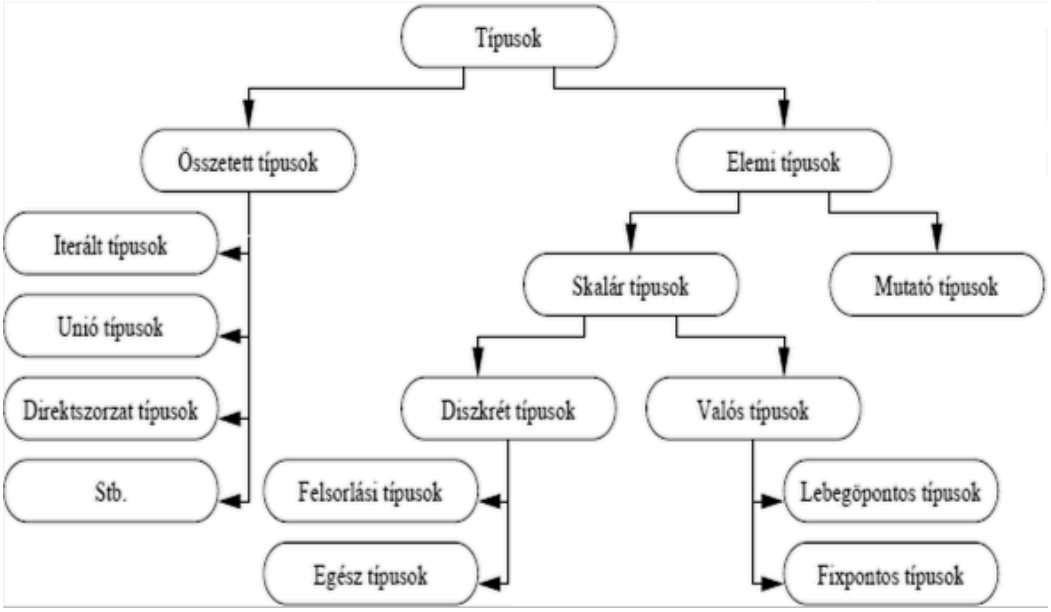
Alaphalmaz: a valós világ minket érdeklő objektumainak halmaza
Specifikációs invariáns: szűkíti az alaphalmazt
Típusműveletek specifikációja

Típusmegvalósítás: „hogyan”

Reprezentációs függvény, típusinvariáns, típusműveletek megvalósítása

Típusosztályok:

Elemi: logikailag felbonthatatlanok
Összetett: meglévő típusokból hozzuk létre



Ada felsorolási típus: `type Hónapok is (Január,...,December);`

ennek címzése: `Hónapok'First, Hónapok'Pos(x), ...`

altípus: a típus értékhalmazának részhalmaza

```
subtype Napok is Integer range 1..31;
```

származtatott típus: átveszi az eredeti tulajdonságait, de nem ekvivalensek

```
type Hosszúság is new Float;
```

Java primitív és referencia típusok

Skalár típusok: diszkrét/valós típusok, értékeik rendezettek.

Felsorolási típusok: a típusértékhalmaz megadható explicit felsorolással, pl. logikai.

Ada Minden felsorolási literál önálló típusérték és van pozíciószáma (0-tól).
A rendezési relációt a felsorolás sorrendje adja.
2 karaktertípus (Character, Wide_Character), Boolean

C++ az *enum* kulcsszóval hozható létre
char, signed char, unsigned char, bool

Java enum, char, boolean

Egész típusok

Ada előjeles egészek: `type Page_Num is range 1..2_000;`
maradékosztályok: `type Byte is mod 256;`
minden egész típus a *root_integer* osztályból származik
Integer, Natural, Positive
műveletek: +, -, *, /, rem, mod, abs, ** *(hatványozás Natural kitevőre)*

C++ short int, int, long int, + unsigned

Java byte (Byte), short (Short), int (Integer), long (Long), char (Character)

Valós típusok: lebegő- és fixpontos

Ada lebegő és fixpontos is lehet, de csak a Float előredefiniált

```
type Real is digits 8;
type Coefficient is digits 10 range -1.0..1.0;
```

minden valós típus a *root_real* típusból származik

C++ float, double, long double

Java float, double
van POSITIVE_INFINITY, NEGATIVE_INFINITY, NaN

Pointer, referencia: olyan objektum, amely megadja egy másik objektum memóriacímét.

Értéke egy memóriacím.

Felhasználás: hatékonyság növeléséhez, dinamikus adatszerkezetek építéséhez, objektumorientált funkciókhoz

Műveletek: dereferencing (a mutatott objektumra hivatkozás), referencing (az objektum címe), értékadás, egyenlőségvizsgálat, új objektum allokálása, deallokálás, néha +,-

Csellengő pointer: kísérlet olyan változó elérésére, ami már nem létezik.

Vannak típus nélküli pointerek? Csak dinamikusan allokált objektumokra mutathat pointer? Mutathat pointer alprogramra?Milyen fajta konstans pointer megengedett? Kötelező önálló nevet adni, vagy csak a mutatott típust kell megadni? Milyen biztonságosan kezelhetők a csellengő pointerek? Mi a megengedett műveletek halmaza? Kapnak a pointerek kezdeti értéket? Lehetséges egy objektumra két pointert állítani?

C++ A legtöbb T típusra a T* a pointer típus.
&: változó címének lekérdezése

Java nincs hagyományos pointer típus: primitív és referencia típusok

5. tétel *Típusok 2.*

Új típus definiálása:

Ada
```
subtype Int is Integer;
type My_Int is new Integer;
```
C++
```
typedef int myint;
```
Java
```
class
```

Tömb: olyan adatszerkezet, amely azonos típusú elemek sorozatait tartalmazza.
Általában egy leképezés egy folytonos diszkrét intervallumról elemek egy halmazára.
Tömbnév(indexértékek) → elem Az alapművelet az indexelés.
Milyen adattípus lehet index, elem? Tartalmazza az indexhatárokat? Mikor dől el a mérete? Ellenőrzi-e az indextúlcsordulást? Van többdimenziós tömb? Van altömbképzés? Van teljes tömbre vonatkozó értékadás? Van tömbkonstans? Megváltoztatható a mérete?

Ada rögzített és megszorítás nélküli indexhatárok is:
```
type A is array(Integer range 2..10) of Boolean;
type Vect is array(Integer range<>) of Integer;
```
az index típusa bármilyen diszkrét típus lehet
az indexhatároknak nem kell statikusnak lennie (futási időben értékeli ki)
megengedett az értékadás azonos típusú tömbök között
pl. string: `type String is array(Positive range<>) of Character;`
attribútumok: A'First, A'Last, A'Range, A'Length

C++ tulajdonképpen pointer: a tömb 0. elemére hivatkozik
nem tudja a méretét

Java méret lekérdezhető, kezdeti érték adható
a karakterek tömbje nem String!

Asszociatív tömb: rendezetlen, a kulcs bármi lehet **C++** **Java**

Direktszorzat (rekord): két típus (S és T) direkt szorzata S × T.
S × T = {(x,y) | x∈S, y∈T}
Az (x,y) komponensek a rekord *mezői*. Az alapművelet a komponenskiválasztás.
Lehet paramétere a típusnak? Van kezdő értékadás? Van teljes rekordra vonatkozó értékadás? Van rekord konstans? Hogyan működik a kiválasztás?

Ada
```
type Complex is record
      Re: Float;
      Im: Float;
end record;
```
változó deklarálása: `C: Complex;`
komponensre hivatkozás: `C.Re  C.Im`
a rekord diszkriminánsa: típusparaméter, pl. `type Szöveg(Hossz:Natural) is`...

C++ struct (érték típus)
inicializálás: *típus* `a = {`*érték1*`,`*érték2*`,...,`*értékn*`}`
(de a konstruktor jobb)

Java nincs

Unió: típusértékhalmaza a komponensek típusértékhalmazának uniója.

Variáns rekord: olyan direktszorzat, amelynek az utolsó komponense unió.

Tag: egyes alkomponensek elérhetősége ennek az értékétől függ.

Megkülönböztetett unió: a rekord tárolja és használja a tagot.

Szabad unió: nincs tag, vagy a rekord nem használja rendeltetésszerűen.

A szabad unió a nyelv típusrendszerét megbízhatatlanná teszi.

Megkülönböztetett unió: hogyan lehet új értéket adni a tag mezőnek?

Ada *case* a rekord létrehozása közben:

```
case Családi_Áll is
      when Házas => Házastárs_Neve: Név;
      when Özvegy => Házastárs_Halála: Dátum;
      when Elvált => Válás_Dátuma: Dátum;
      when Egyedülálló => null;
end case;
```

megtörése esetén futás idejű hiba

C++ szabad unió

Java nincs unió, helyette osztályok és öröklődés használata javasolt

Halmaz: speciális iterált típus

Mi lehet az eleme? Hány eleme lehet? Megvannak a hagyományos halmazműveletek? Mikor ekvivalensek? A mezőneveket figyelembe vesszük, vagy csak a struktúrát? Számít a mezők sorrendje? Tömböknél elég a számosság egyezése, vagy a határoknak is egyezniük kell?Értékül adhatóak egymásnak? Van-e és hogyan működik az automatikus, identitás-, bővítő, szűkítő, toString konverzió?

Java identitáskonverzió: csak boolean

bővítő konverzió: mindenről minden, nála bővebbre

szűkítő konverzió: mindenről minden, nála szűkebbre

6. tétel *Absztrakt adattípusok a programozási nyelvekben*

Egy szoftverrendszer mindig végrehajt

bizonyos **tevékenységeket**

bizonyos **adatokon**.

A tervezés központi kérdése, hogy a rendszer felépítését a kettő közül melyikre alapozzuk.

Procedurális absztrakció: top-down megoldás: feladat → alfeladat → ...

Hátránya: nehéz a specifikáció változásaihoz adaptálni.

Az alprogram specifikációja egy magasabb szintű specifikációtól függ, így egy kis változásnak is nagyon nagy hatása lehet.

Adatabsztrakció: alulról felfelé építkezés

A rendszer hosszú távon jobban jellemezhető az objektumaival, mint a tevékenységeivel. Tehát először jellemezzük az objektumokat és az objektumok osztályait (adattípusokat), majd az újabb adattípusokat a már meglevők felhasználásával tervezzük.

Elvárások a programozási nyelvekkel szemben:

- Modularitás: az egyes típusokat önálló fordítási egységekben lehessen megvalósítani

- Adatrejtés: a nyelv támogassa a reprezentáció elrejtését
  Biztosítja, hogy a típus használója csak a specifikációban megadott tulajdonságokat használhassa ki.
- Konzisztens használhatóság: a felhasználói és a beépített típusok ne különbözzenek a használat szempontjából
- Generikus programsémák támogatása: a programozó minél általánosabban írhassa meg a programjait
- Specifikáció és implementáció szétválasztása külön fordítási egységbe
- Modulfüggőség kezelése: a fordítóprogram maga kezelje a modulok közötti relációkat

Absztrakt típusok megvalósításának eszközei:

Saját adattípus megvalósítható önálló fordítási egységként? Szétválasztható a specifikáció és az implementáció? Megvalósítható a reprezentáció elrejtése? Milyen láthatósági szintek vannak? Van-e azonosító- és operátortúlterhelés? Ugyanúgy használhatóak a saját típusok, mint a beépítettek?

- Modularitás: egyre inkább egy modul – egy típus elvet jelent
  Kivétel: „alprogram könyvtárak”, implementációs célokat szolgáló típusok
- Adatrejtés: leggyakrabban háromszintű:
  nyilvános (public), *védett* (protected), *privát* (private)
  Típus vagy objektum szinten szabályoz?
- Konzisztens használhatóság: túlterhelés, átlapolás
- Specifikáció és implementáció szétválasztása külön fordítási egységbe
  - **Ada** A specifikáció és a reprezentáció egy fordítási egységbe kerül.
    Így a reprez. fizikailag nincs elrejtve, de az így megszerzett információkat a nyelv nem engedi kihasználni.
  - **C++** A specifikációt fizikailag be kell másolni minden olyan fordítási egységbe, ahol az adott típust használni szeretnénk → nem valódi szétválasztás.
    (A header fájlt ennek megkönnyítésére használjuk: az előfordító másolja be.)
  - **Java** A specifikáció és az implementáció összemosódik.
    Segítséget a dokumentációs lehetőségek jelentenek (Javadoc).
- Modulfüggőség kezelése
  - **Ada** Függőségek megadása: *with* kulcsszóval.
    Ez garantálja, hogy azoknak a moduloknak a spec. része is fordításra kerüljön.
  - **C++** Ezt a feladatot teljesen a programozóra bízza, minden egységet önállóan kezel.
    Speciális eszköz könnyítésként: *make*

7. tétel *Alprogramok*

Absztrakció a programozásban:

Lényeges: mit csinál?
Lényegtelen: hogyan van implementálva?

Van eljárás? Van függvény? Milyen paraméterátadási módok léteznek? Meghatározható az információátadás iránya? Adható a formális paramétereknek alapértelmezett érték? Túlterhelhetők az

alprogramnevek? Átlapolhatók az operátorok? Definiálhatók új operátorok? Az alprogram típus? Lehet alprogrammal paraméterezni?

Függvényabsztrakció: egy kiszámítandó kifejezést tartalmaz

Ada
```
function Kerulet(R: Float) return Float is
begin
      return 2.0 * R * 3.14;
end;
```

Eljárásabsztrakció: egy végrehajtandó parancsot tartalmaz

Absztrakciós mechanizmus: az a nyelvi konstrukció, ami megengedi, hogy a programozó megragadja az absztrakciót és a program részeként reprezentálja, *pl. eljárások és függvények.*

Függvény: `function I(FP1;...;FPn) return T is E`

Egy I azonosítót kapcsol egy adott függvényabsztrakcióhoz.
A függv. absztrakció megfelelő paraméterekkel híváskor eredményt ad vissza.
Felhasználói szemlélet: leképezés az argumentumok és az eredmény között.
Megvalósítói szemlélet: a függvénytörzs kiértékelése.
Végrehajtása: kifejezés értékének meghatározása
Neve: főnév vagy melléknév *(pl. Elemek_Száma)*

Eljárás: `procedure I(FP1;...;FPn) B`

Felhasználói szemlélet: érzékeljük a körülmények változását
Megvalósítói szemlélet: a kódolt algoritmus
Végrehajtása: utasítás
Neve: ige *(pl. Egyenest_Rajzol)*

Tetszőleges szintaktikai osztály fölött létrehozhatunk absztrakciókat feltéve, hogy ez valamifajta számítást specifikál, *pl. absztrakt adattípusok, generic*.

Alprogram: programegység, amelynek nevet adunk és amelynek viselkedését paraméterezzük.

Végrehajtás kezdeményezése: meghívással
A program darabolásának eszköze.
Célja: számítások elkülönítése egymástól, újrafelhasználhatóság
Definíciója: alprogram = (név, paraméterek, környezet, törzs)

Formális paraméterekkel írjuk le az adatcsere elemeit,
a használatkor az *aktuális paraméterek* kerülnek ezek helyére (paraméterátadás)

Információáramlás iránya:

Input:	hívó → alprogram
Output:	hívó ← alprogram
Update:	hívó ↔ alprogram

Paraméterátadási technikák:

1. Érték szerinti: a formális paraméter az alprogram egy lokális változója

C++
```
int lnko(int a,int b) {
      while (a != b)
            if (a > b) a-=b;
            else b-=a;
```

```
        return a;
    }
```

- ennek az aktuális paraméter adja a kezdőértéket
- az aktuális paramétert egyszer, a végrehajtás előtt értékeli ki
- a futás közben az aktuális és a formális paraméter nincs hatással egymásra

2. Cím szerinti: az aktuális paraméter egy változó vagy egy változó komponensét meghatározó kifejezés
   - aktuális paraméter kiértékelése: a hozzá rendelt memóriaterület címének meghatározása
   - az aktuális paramétert a végrehajtás előtt értékeli ki
   - az alprogramban hivatkozhatunk a formális paraméter értékére és módosíthatjuk is
3. Eredmény szerinti: az alprogram a formális paraméterben kiszámított eredményt visszahelyezi az aktuális paraméterbe.
   - a formális paraméter az alprogram lokális változója
   - az alprogram végén kimásolódik az aktuális paraméterbe
   - nem kapja meg az aktuális paraméter értékét híváskor → output
   - érték/eredmény szerinti: megegyezik az érték szerintivel, de végén az aktuális paraméter felveszi a formális pillanatnyi értékét
4. Név szerinti: az aktuális paraméterként leírt teljes kifejezés átadódik és minden használatkor dinamikusan kiértékelődik, *pl. a[i]*.

Formális-aktuális paraméterek megfeleltetése: `procedure Get_Line(File F, Item S, Last N)`

Pozícionális formában: a specifikáció szerinti sorrendben kell felsorolni

```
Get_Line(F,S,N);
```

Névvel jelölt formában: a paraméterek sorrendje tetszőleges

```
Get_Line(File=>F,Last=>L,Item=>S);
```

Kevert formában: a pozícionálisan megadott paramétereknek kell elöl állniuk

```
Get_Line(F,Last=>L,Item=>S);
```

Túlterhelés: azonos név, paraméterek száma/típusa különböző

A hívásból fog kiderülni, hogy melyikre gondoltunk.

Rekurzió: közvetlenül vagy közvetve önmagát hívó alprogram.

A cím szerinti átadás veszélyes lehet, mert a sorozatos hívások „összeakadhatnak” egymással.

Ada összetett típusú értékek esetén a fordító választ az érték-eredmény és a cím szerinti átadás között
a függvényeknek csak *in* paraméterei lehetnek

C++ csak függvény van, eljárás: *void* visszatérési érték
csak érték szerinti paraméterátvétel, cím szerintihez mutatók/referencia kell
lehet a paramétereknek alapértelmezett értéket adni
túlterhelés lehetséges

Java csak valamelyik osztály metódusa lehet
paraméterátadás: primitív típusokat érték szerint, objektumokat referencia szerint

8. tétel *Kivételkezelés*

Cél: jó minőségű program (helyes, robosztus/megbízható)

Elvárások:

- hibák megkülönböztetése (fellépés helyén)
- a kezelő kód különüljön el a tényleges kódtól → *karbantarthatóság*
- a hiba könnyen jusson el oda, ahol kezelni kell (nem biztos, hogy a kiváltó szint le tudja kezelni)
- kötelező legyen kijavítani a hibákat

Ad hiba- vagy kivételkezelést a nyelv? Hogyan kell a kivételeket definiálni, kiváltani, kezelni? Milyen a szintaktikai formájuk? Lehet kivételosztályokat képezni? Paraméterezhető a kivétel? Létezik finally *tevékenység? Ha nem, szimulálható? Biztosít olyan konstrukciót, ahol nem kell explicit megadni, hogy mely kivételt kell továbbadni? Újrakezdhető az alprogram a hiba kezelése után? Megadható, hogy az alprogram milyen kivételeket dobhat? Párhuzamos környezetben vannak speciális kivételek? Mi történik a váratlan kivételekkel? Kiváltható kivétel másik kivétel kezelése közben? Melyik kivételkezelő kezeli a kivételt?*

C++ Kivétel lehet egy tetszőleges típusú objektum.

throw utasítás objektum nélkül = *terminate* hívása

Egy *catch* ág elkapja a kivételt, ha az egyik teljesül:

a két típus pont ugyanaz,

a *catch* ág típusa publikus bázisosztálya az eldobott objektumnak,

a *catch* ág típusa mutató, az objektum mutató és mutatókonverzióval egymásba konvertálhatók.

`catch (...)` : az összes kivételt elkapja

`int f(int x) throw(A,B,C)` : csak A, B és C kivételt dobhat és a leszármazottaikat

`int f(int x) throw()` : nem válthat ki kivételt

Java hasonló a C++-hoz, de van *finally*

kiváltható kivételek definiálása a *throws* kulcsszóval

többszálú program: *uncaughtException()* függvény fut le

minden kivétel a *java.lang.Throwable* leszármazottja

ellenőrzött: *Exception* leszármazottai, számítani lehet rájuk

ellenőrizetlen: *Error* leszármazottai, nem lehet számítani rájuk (OutOfMemoryError)

Eiffel Újrakezdés: alternatív megoldás a szerződés betartatására

```
rescue
        utasítások;
        retry;
```

Szervezett pánik: ha nincs rá mód, hogy betartsuk a szerződést

elfogadható állapotba hozza az objektumokat és jelzi a kudarcot a felhasználónak

```
rescue
        reset(arg);
```

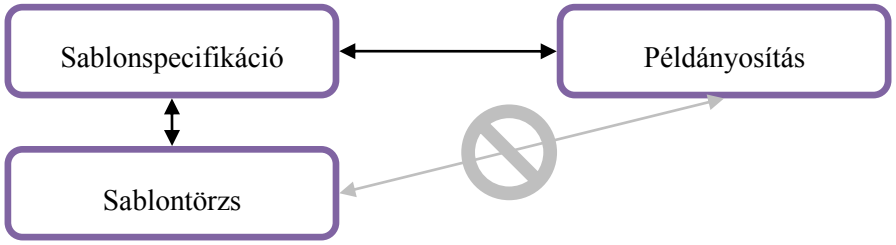
9. tétel *Sablonok*

újrafelhasználható, könnyen karbantartható program →

minél általánosabb típusok és alprogramok

elemek típusától függetlenül tervezünk

Sablonszerződés modellje:



Milyen nyelvi elemekből tudunk sablonokat létrehozni? (Új adattípusból, alprogramból?) Milyen paraméterei lehetnek? (Típusok, objektumok, alprogramok?) Létrehozható paraméter nélküli sablon? Felhasználó által definiált típus is lehet aktuális típusparaméter? Adhatók megszorítások a formális generic paramétereknek? Egymásba ágyazhatók? A példányosítás fordítási (statikusan) vagy futási (dinamikusan) időben történik?

Java

```
public class Box<T> {
        public <U> void inspect(U u) { ...
```

pl. T: *java.lang.Integer*, U: *java.lang.String*

megszorított típusparaméter: `<U extends Valami>`

interfésszel: `<U extends Valami & MyInterface>`

C++

template-ek: minimális szintaxisellenőrzés, egyébként az aktuális típussal helyettesít

nem írható elő bizonyos műveletek megléte → az ilyen hibák példányosítások jelentkeznek

a template példányosítása teljesen automatikusan történik

```
template <typename T>
```

Ada

lehet megadott őstípusból származó típussal paraméterezni,

kiköthető, hogy az aktuális paraméter valamilyen típusosztályba tartozzon

megadhatók a használni kívánt további műveletek

```
generic
        type Item is private;
        type Index is (<>);
        type Vector is array(Index range<>) of Item;
        with function "<"(X,Y:Item) return Boolean is <>;
```

Eiffel

formális paraméterei osztályok lehetnek

megadható, hogy az aktuális paraméternek mely osztályból kell származnia

Előny: a használat helyessége a sablon megírásakor ellenőrizhető

Hátrány: a szükséges közös tulajdonságokat jóelőre tudni kell

```
class ARRAY[G]
class TREE[INTEGER]
```

korlátozott generic: pl. a kulcsot megszorítjuk a *HASHABLE* osztállyal

```
        class HASH TABLE[G,KEY->HASHABLE]
```

10. tétel *Helyességbiztosítást támogató nyelvi eszközök*

Szerződés a felhasználó és az implementáló között:

Előfeltétel: a feltétel, ami szükséges a feladat helyes működéséhez

Utófeltétel: a feltétel, amit a függvény teljesít a helyes végrehajtás után

Ha a felhasználó teljesíti az előfeltételt, akkor a függvény lefut, és a futás végén az utófeltétel igaz lesz.

Hoare-hármas szintaxisa: {Q} S {R} Q és R állítások, S program

Ha kiinduláskor igaz Q és végrehajtjuk S-t, akkor S terminálásakor R igaz lesz.

pl. {true} x := 5 {x=5}
{x=y} x:=x+3 {x=y+3}
{false} x := 3 {x=8}

Ha {Q} S {R} és ∀R'-re, amire {Q} S {R'} igaz, hogy R ⇒ R', akkor
R az S Q-ra vonatkozó **legerősebb utófeltétele**.

Ha {Q} S {R} és ∀Q'-re, amire {Q'} S {R} igaz, hogy Q' ⇒ Q, akkor
Q az S R-re vonatkozó **leggyengébb előfeltétele**. Jele: wp(S,R).

Tulajdonságai:

1) A csoda kizárásának törvénye: wp(S,false) = false
2) Monotonitás: ha Q ⇒ R, akkor wp(S,Q) ⇒ wp(S,R)
3) Linearitás: wp(S,P **and** Q) = wp(S,P) **and** wp(S,Q)
4) wp(S,P) **or** wp(S,Q) = wp(S,P **or** Q)

Konzisztencia: C osztály konzisztens, ha:

1) minden *p* konstruktorára: $\{pre_p\}$ p $\{INV_c\}$
   (Minden konstruktor teljesíti a saját előfeltételét és az objektum invariánsát igazzá teszi.)
2) minden *r* metódusára: $\{pre_r$ and $INV_c\}$ r $\{post_r$ and $INV_c\}$
   (Minden metódus teljesíti saját elő- és utófeltételeit, miközben az invariáns igaz marad.)

Ciklushelyesség: C osztály r metódusa a.cs.a. ciklushelyes, ha minden metódusra:

1) {True} INIT {INV} (Az inicializáló igazzá teszi az invariánst.)
2) {True} INIT {VAR ≥ 0} (Az inicializáló a ciklusváltozót ≥0 értékre állítja.)
3) {INV and then not EXIT} BODY {INV}
   (Ha teljesül az invariáns és nem lépünk ki, akkor az iteráció után is teljesülni fog.)
4) {INV and then not EXIT and then VAR = v} BODY {0 ≤ VAR < v}
   (Ha teljesül az invariáns, nem lépünk ki és a ciklusváltozó értéke *v*, akkor az iteráció után a változó 0 és *v* között lesz.)

Kivételhelyesség: C osztály r metódusa a.cs.a. kivételhelyes, ha *rescue* blokkjának minden ágára:

1) Ha b *retry*-jal kezdődik: {True} b $\{INV_C$ and $pre_r\}$
   (*b* lefutása után az invariáns és *r* előfeltétele is teljesülni fog.)
2) Ha b nem *retry*-jal kezdődik: {True} b $\{INV_C\}$
   (*b* lefutása után az invariáns teljesülni fog, de a *retry* hiánya miatt *r* előfeltétele nem biztos.)

Megadhatunk az alprogramoknak elő- és utófeltételeket? Milyen formában? Megadhatunk specifikációs és típusinvariánst? Képes ciklushelyesség, kivételhelyesség ellenőrzésére? Milyen az öröklődéskor az elő- és utófeltételek és a típusinvariánsok kapcsolata?

Alphard Cél: a Hoare-féle helyességbizonyításhoz lett volna egy specifikációs eszköz.

Típusok: formok

```
form típusnév (formális paraméterek) =
      beginform
      specifications
            requires: formális paraméterek megszorításai
            let típus leírására használt absztrakt adattípus
            invariant típusinvariáns
            initially absztrakt kezdeti objektum tulajdonságai
            function művelet1(par1:típ1,par2:típ2)
                  pre előfeltételek
```

```
                post utófeltételek
                ...
        representation
                ...
        implementation
                ...
        endform;
```

reprezentációs függvény: visszaképez az abszrakt adattípusra

Eiffel programozás szerződéssel, jogok és kötelezettségek

```
forth is
        require
                not after: not after
        do
                léptetés
        ensure
                position = old position + 1
end forth
```

Kulcsszavak: `invariant, variant`

old operátor: a végrehajtandó metódusba lépés előtti állapot, pl. `x = old x + 1`

Az elő- és utófeltételek újradeklarálhatóak öröklésekor: *require else*, *ensure then*

megtartja vagy gyengíti az előfeltételt, nem tesz hozzá újat

megtartja vagy szűkíti az utófeltételt

tetszőleges állítás, amit a futtató ellenőriz: *check*

`no` : nem ellenőrzi az állításokat

`require` : az előfeltételeket ellenőrzi

`ensure` : elő- és utófeltételeket is ellenőrzi

`invariant` : előzőeket + osztályinvariánst is

`loop` : előzőeket + ciklusokat is

`all` : előzőeket + *check* utasításokat is

D assert: egyszerű feltétel; ha nem teljesül, AssertExceptiont dob

elő- és utófeltételek: `in {...} out{...} body{...}`

in: híváskor fut le, *out:* blokk elhagyásakor fut le (bármilyen módon hagyjuk is el)

Szabály: *in* és *out* nem változtathatja meg a környezetet (jellemzően assertek)

osztály invariáns: speciális tagfüggvény: invariant()

adattagokat nem módosíthat

konstruktor után, destruktor előtt, tagfüggvények hívása előtt és után lefut

az objektum helyességét ellenőrzi

11. tétel *Objektumorientált programozás 1.*

Modellezési alapelvek:

Absztrakció: a valós világot leegyszerűsítjük, és csak a cél szempontjából lényeges részekre összpontosítunk.

Megkülönböztetés: az objektumokat a számunkra lényeges tulajdonságaik, viselkedési módjuk alapján megkülönböztetjük.

Osztályozás: az objektumokat kategóriákba, osztályokba soroljuk, ezek pedig hordozzák a hozzájuk tartozó objektumok jellemzőit (az objektumok mintái).

Általánosítás: hasonlóságokat, különbségeket keresünk, ezek alapján osztályokba rendezünk.

Kapcsolatok felépítése, részekre bontás: pl. ismeretségi, tartalmazási kapcsolat

Objektum:

- információt tárol, kérésre feladatokat hajt végre
- belső állapota van, üzeneten keresztül lehet megszólítani
- adattagok + műveletek
- két objektum azonossága független a tárolt értékektől
  (akkor sem azonos, ha állapotaik megegyeznek!)
- inicializálását a konstruktor végzi

Egy objektumorientált program egymással kommunikáló objektumok összessége.

Osztály: olyan objektumminta vagy típus, mely alapján példányokat hozhatunk létre.
osztályváltozó, osztálymetódus

Példány: az osztályból képzett objektum
példányváltozó, példánymetódus

Az objektumok műveletei:

- **Export**
  - Létrehozó (konstruktor)
  - Állapotmegváltoztató, *pl. pop, push*
  - Szelektor: kiemeli az objektum egy bizonyos részét, *pl. access*
  - Kiértékelő: lekérdezi az objektum jellemzőit, *pl. size*
  - Iterátor: bejáráshoz

Kliens-szerver kapcsolat

Kliens: aktív objektum: másik objektumon végez műveletet. Nincs export felülete.
Szerver: passzív objektum: másoktól érkező üzenetekre vár. Nincs import felülete.
Ágens: általános objektum. Export és import felülete is van.

this, Self: a metódus hívója

Társítási kapcsolatok:

1) **Objektumok között**
   - ismertségi *(használati)*
   - tartalmazási *(egész-rész)*
     - gyenge *(a rész kivehető az egészből)*
     - erős *(a rész nem vehető ki az egészből)*
     - az egész objektum mindig ismeri a részét!
2) **Osztályok között**
   - ismertségi vagy tartalmazási
   - multiplicitás: egy-egy, egy-sok, sok-sok
   - kötelező vagy opcionális

12. tétel *Objektumorientált programozás 2.*

Öröklődés

Alapgondolat: a gyerekek öröklik őseik metódusait és változóit.
A gyerek minden új metódusa és változója hozzáadódik az örököltekhez.

Polimorfizmus (többalakúság): egy változó nem csak egyfajta típusú objektumra hivatkozhat.
Statikus típus: deklaráció során kapja
Dinamikus típus: futás közben kapja, a statikus típus vagy annak leszármazottja

Dinamikus kötés: az a jelenség, hogy a változó aktuális dinamikus típusának megfelelő metódusimplementáció hajtódik végre.

Java minden felüldefiniálható, ami nem *final*

C++ a *virtual* kulcsszóval engedi meg a felüldefiniálást

Eiffel az altípus a *redefine* záradékkal jelzi a felüldefiniálást

SmallTalk Az előzőekkel ellentétben dinamikus kötésnél „látja” a dinamikus típus metódusait is

13. tétel *Objektumorientált programozás 3.*

Többszörös öröklődés: több szülő

Java nem engedi: interface használata ajánlott
előny: biztonságos, *hátrány:* korlátozza az újrahasznosítást

Eiffel explicit átnevezés vagy elrejtés **kell**

C++ explicit *scope* operátor kell

Akkor biztonságos, ha:
az altípus metódusai megőrzik az őstípusok viselkedését:
előfeltétel: kontravariancia (az altípus gyengébb)
utófeltétel: kovariancia (az altípus erősebb)
az altípusok megőrzik az őstípusok tulajdonságait → típusinvariánsukra: kovariancia

C++ az altípus kovariáns metódusai nem átdefiniálnak, hanem túlterhelnek

Eiffel be lehet vezetni kovariáns metódusokat az altípusban, de nem típusbiztos!

Problémák: *Egyező adattagok hányszor jelenjenek meg? Melyik metódus érvényes?*
Megoldási módszerek:
A fordító hibával leáll, ha kétértelműséget talál. (ambiguous)
A származtatott osztály mondja meg, hogy melyiket szeretné használni.
Az ősosztály mondja meg, hogy mit szeretne tenni ilyen esetben.

C++ A-nak a B és C virtuális bázisosztályának kell lennie
minden adattag csak egyszer öröklődik

```
class D : public B, public C {
      public:
            using B::a;
            using C::f;
};
```

Eiffel Átnevezés (rename):

```
class D inherit
        C
        rename a as ac
        end
        ...
end
```

Kiválasztás (select):

```
class B inherit
        A
        redefine
                f...
        end;
feature
        f...
end;
```

Absztrakt osztály: felső szinten fogja össze a közös tulajdonságokat.
Van olyan metódus, aminek csak specifikációja van, törzse nincs.

Interfész: teljesen absztrakt osztály. Metódusok egy csoportját specifikálja a törzsük implementálása nélkül. (konstans változók)
Többszörös öröklődés engedélyezett.

Objektumok – összefoglalás programozási nyelvenként

Van öröklődés? Van többszörös öröklődés? Hogyan van megoldva az adattagok és a metódusok elrejtése? Támogatja a polimorfizmust és a dinamikus összekapcsolást? Van alapértelmezett ősosztály (pl. Object)? Van absztrakt osztály? Konstruktor, destruktor, Garbage Collector, standard objektumkönyvtárak, osztályszintű attribútumok és metódusok?

SmallTalk minden objektum, de a változók típusnélküliek
szabványos objektumkönyvtárak
virtual machine, garbage collector
nincs többszörös öröklődés
láthatóság: minden belső változó protected, minden metódus public
van osztályszintű attribútum és metódus
közös ősosztály: Object
nincs destruktor, nincs absztrakt osztály

C++ nincs Garbage Collector
adattagok és metódusok elrejtése megoldott: public, protected, private
többszörös öröklődés, absztrakt osztály
polimorfizmus, dinamikus kötés csak mutatók és referenciák esetén
virtual: dinamikus kötés támogatására
a leszármazott nem örököl konstruktort, destruktort, értékadás operátort, barátokat
a barátok elérhetik a private mezőket
osztályváltozó, ~metódus: static

Java hasonló a **SmallTalk**hoz
többszörös öröklődés helyett interfész
a primitív típusoknak van csomagoló osztályuk, pl. boolean → Boolean
final: tiltja az örököltetést / felüldefiniálást / érték megváltoztatását

abstract: nem példányosítható
nincs operátortúlterhelés
nincs destruktor, csak finalize()

Eiffel többszörös öröklődés
van absztrakt osztály és metódus
a leszármazott mondja meg, hogyan szeretné felhasználni az örökölt elemeket
átnevezés, export státusz megváltoztatása, metódus felüldefiniálása, műveletek absztrakttá tétele, ismételt öröklődés

14. tétel *Párhuzamosság*

Időosztásos rendszer

Folyamat: olyan műveletsor, amelyet egy szekvenciálisan végrehajtott program valósít meg.
A programból, a be- és kimenő adatokból és egy állapotból áll.
5 állapota lehet:
futó: a CPU éppen őt futtatja
kész (ready): futtatható, éppen várakozik, hogy sorra kerüljön
blokkolt: várakozik egy külső eseményre
megszakított
halott: befejeződött
Hagyományos folyamatban: vezérlő szál, utasításmutató

Szál: hasonlít egy folyamathoz, különbség:
A folyamat kernel szintű fogalom:
minden egyede speciális tulajdonságokkal rendelkezik,
más folyamatok csak rendszerhívásokon keresztül férnek hozzá,
a folyamatleíró struktúra a kernelben van, a felhasználói program nem tudja elérni,
a program eljárásai, függvényei felhasználói területen vannak → elérhetőek
A szál felhasználói szintű fogalom:
a szálleíró struktúra a felhasználói területen van, közvetlenül elérhető,
rendelkeznek regiszter- és állapotleíró adatokkal,
egy folyamat minden szála ugyanazt látja

Kommunikáció a folyamatok között:
Osztott változókkal: a program változóinak egy része hozzáférhető mindkét folyamat számára.
Fontos a folyamatok összehangolása, nehogy egyidőben írjanak!
Explicit üzenetküldéssel: aszinkron, szinkron
Szinkronizáció: a két folyamat vezérlését kapcsolatba hozza

Viselkedésük alapján lehetnek:
egymástól függetlenek,
egymással versengők (versengenek egy kizárólagosan használható erőforrásért),
egymással együttműködők (ugyanazon probléma különböző részein dolgoznak)

Multiprogramozott rendszer: szekvenciális programok halmaza, amelyek üzenetek és szinkronizációs jelek segítségével együttműködnek egymással és versengenek a korlátozott erőforrásokért.

Kritikus szakasz: a program azon szakasza, ahol a közös adatterületet használó folyamatok ezt az adatterületet módosítják. Megköveteljük, hogy:

a kritikus szakasz véges idő alatt befejeződjön,
egy program véges idő alatt beléphessen a kritikus szakaszba,
ha egy program a kritikus szakaszon kívül leáll, az ne befolyásolja a többit

Kölcsönös kizárás: egyszerre csak egy program lehet a kritikus szakaszban.

Kommunikációs modellek:

1) **Szinkron kommunikáció (randevú)**

Ha egy folyamat kommunikálni akar egy másikkal, akkor a végrehajtása felfüggesztődik a másik válaszáig. A kérő folyamat a randevú kódjának lefutása után futhat tovább.
Az információ áramlása kétirányú lehet.

Ada belépési pontot kell definiálni (*entry*), amihez kódot rendelünk (*accept*)
egy másik szálról meghívhatjuk a belépési pontot

```
task HF is                          task Diák;
      entry Bead;
end HF;
task body HF is                     task body Diák is
begin                               begin
      accept Bead;                        HF.Bead;
end HF;                             end Diák;
```

Aszimmetrikus: megkülönböztetjük a hívót és a hívottat
a hívó ismeri a hívottat, fordítva nem
csak egy randevúra vonatkozó szerepek
Szinkron: akkor jön létre, amikor mindketten akarják
Pont-pont kapcsolat: mindig 2 szál vesz részt benne
A fogadó folyamat hajtja végre az *accept*et, törzse is lehet.
Kommunikáció: az *entry* paramétereivel
lehetnek: *in, out, in out*

2) **Aszinkron kommunikáció üzenetekkel**

Ha egy folyamat üzenetet küld, akkor az a fogadó folyamatnál egy üzenetsorba kerül, a küldő folyamat fut tovább. A fogadó üzenetnek a feldolgozáshoz ki kell vennie az üzenetet az üzenetsorából.
Közös változók: a folyamatok ugyanazt a memóriarészt látják → probléma adódhat
Kommunikáció: kellenek kölcsönös kizárást garantáló eszközök. Módszerek:
nyelvi mechanizmus az adatok védelmére (szemaforok, monitorok)
a folyamatok üzenetekkel kommunikálnak
Prioritások, késleltetések, időzítések...

Szemafor: egy egész értékű számláló és a hozzá tartozó várakozási sor.

Két művelet:

P passeren (áthaladni) → wait
V vrijmaken (szabaddá tenni) → signal

wait(S):
ha S > 0, akkor S := S – 1,
különben a folyamat blokkolódik és a várakozási sorba kerül, amíg fel nem ébresztik
signal(S):
ha van várakozó folyamat a sorban, akkor azt felébreszti
egyébként S := S + 1

Monitor: az osztott adatokról ad információt

Szerkezete:
```
monitor_név
begin
      lokális adatok
      eljárások
      lokális adatok értékadásai
end név;
```

Biztosítja, hogy csak egy folyamat hajthat végre egy monitoreljárást egy adott időben. Mielőtt egy másik folyamat belép, az előzőnek véget kell érnie.

Egyéb nyelvi elemek:

cobegin-coend: párhuzamos folyamat kezdeti és végpontja
select: feltételhez kötött párhuzamos indítás
wait: folyamatot késleltet
fork: párhuzamos folyamat elindítását váltja ki
join: egyesíti a párhuzamos folyamatokat
quit: befejez egy folyamatot

Select

1) **Többágú hívásfogadás**
   a hívott választhat egy hívót valamelyik várakozási sorból
```
select
      accept Művelet_1(...) do ... end;
or
      accept Művelet_2(...) do ... end;
...
end select;
```
   Végtelen folyamat: ha ezt egy végtelen ciklusba tesszük
   Terminál tatás: ha nem akarhatja használni senki, akkor kilép
   `or terminate;` az `end select;` elé
2) **Feltételhez kötött hívásfogadás**
   A select egyes ágaihoz feltétel rendelhető.
```
or
      when feltétel => accept E2 do ... end;
```
   Zárt az ág, ha a feltétele hamis. A többi ág *nyitott*.
3) **Időhöz kötött hívásfogadás**
   Ha megadott ideig nem érkezik hívó, akkor terminálunk.
```
or
      delay 3.0;
```
4) **Feltételes hívásfogadás**
   Ha most azonnal egy hívó sem akar vele randevúzni, akkor nem vár, hanem fut tovább.
```
select – or – else – end select;
```
5) **Időhöz kötött hívás:** egy ideig vár a randevúra, itt a hívóban van *or delay* a *select*ben
6) **Feltételes hívás:** a hívóban szerepel az *else* ág a *select*ben