



Eötvös Loránd Tudományegyetem
Informatikai Kar

Alkalmazott modul: Programozás

1. fejezet

Szoftverfejlesztés, programozási paradigmák

Giachetta Roberto

A jegyzet az ELTE Informatikai Karának 2015. évi
Jegyzetpályázatának támogatásával készült

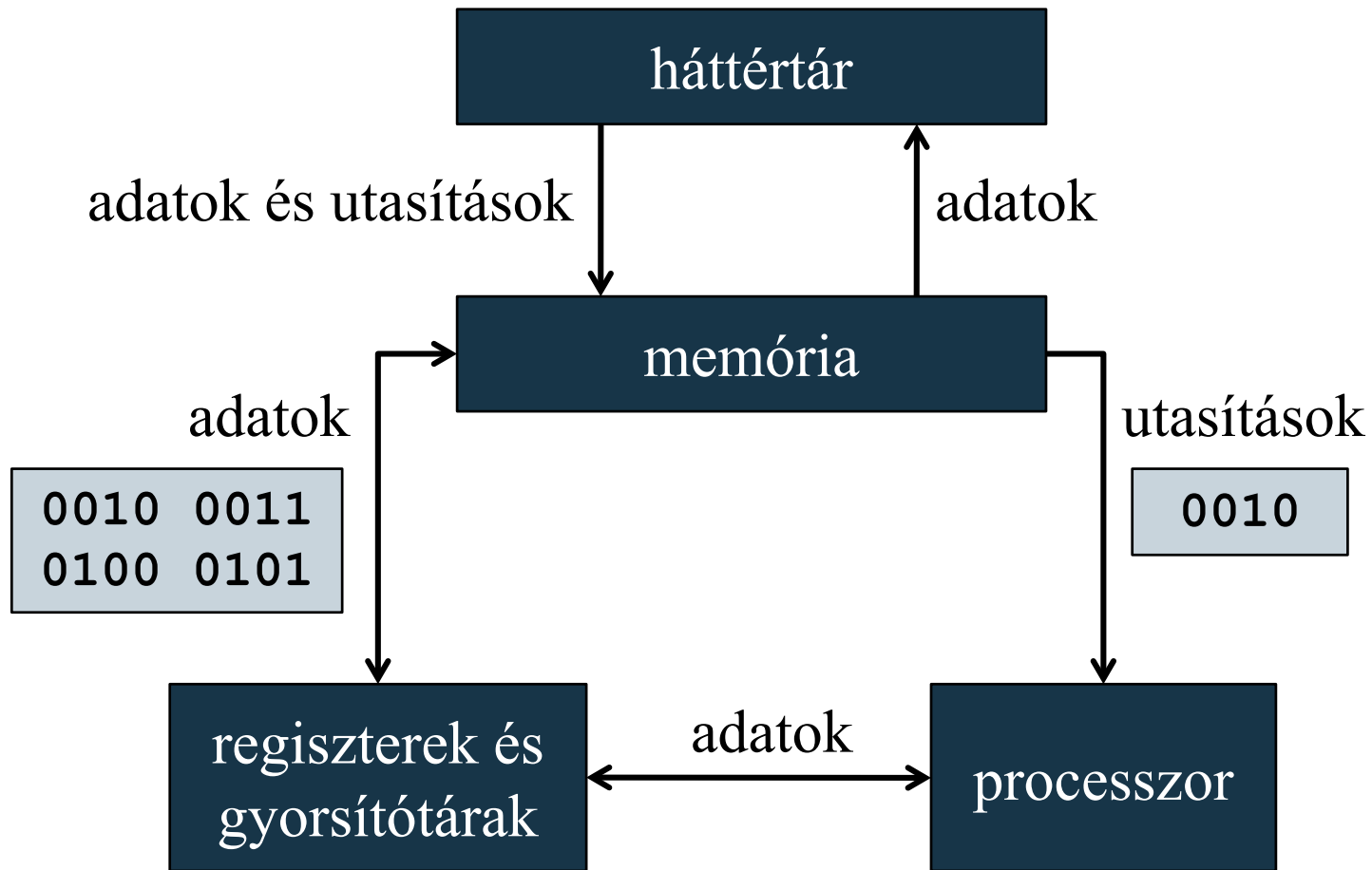
Szoftverfejlesztés

A program

- *A program*
 - *matematikailag*: állapotterek (értékek direktszorzata) felett értelmezett reláció
 - *informatikailag*: utasítások sorozata, amelyek műveleteket hajtanak végre a megadott értékekkel, az *adatokkal*
- A programban foglalt utasítássorozatot, vagy *programkódot* a *processzor* (CPU, GPU, ...) hajtja végre
 - a processzor korlátozott utasításkészlettel rendelkezik, ezért összetett utasításokat nem képes véghezvinni
 - a végrehajtáshoz segéd táraikat (regiszterek, gyorsítótárok) használ, és kommunikál a *memóriával*

Szoftverfejlesztés

A program



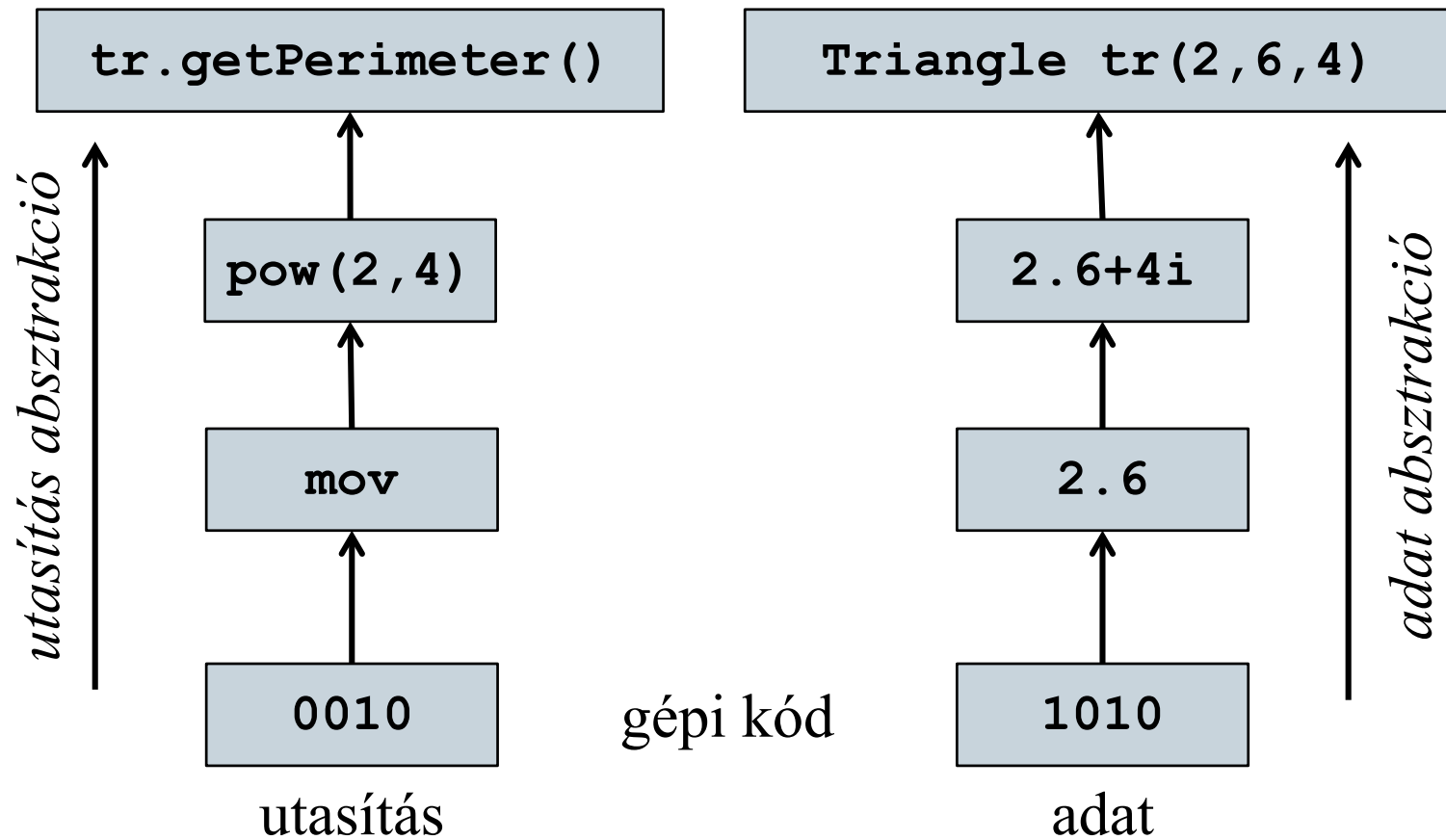
Szoftverfejlesztés

Absztrakció a szoftverfejlesztésben

- A processzor által értelmezhető utasításkészletet és adathalmazt nevezzük *gépi kódnak* (*object code*)
- A programokat ennél magasabb szinten kell elkészíteni, ezért szükségünk van a működés és az adatkezelés absztrakciójára:
 - az *utasításabsztrakció* (*control abstraction*) biztosítja, hogy a processzor egyszerű, egymást követő utasításai (összeadás, mozgatás, összehasonlítás) mellett összetett parancsokat és vezérlési módszert fogalmazzunk meg
 - az *adatabsztrakció* (*data abstraction*) lehetővé teszi, hogy különböző típusokba soroljuk adatainkat, amelyek meghatározzák az értéktartományt, és az elvégezhető műveleteket

Szoftverfejlesztés

Absztrakció a szoftverfejlesztésben

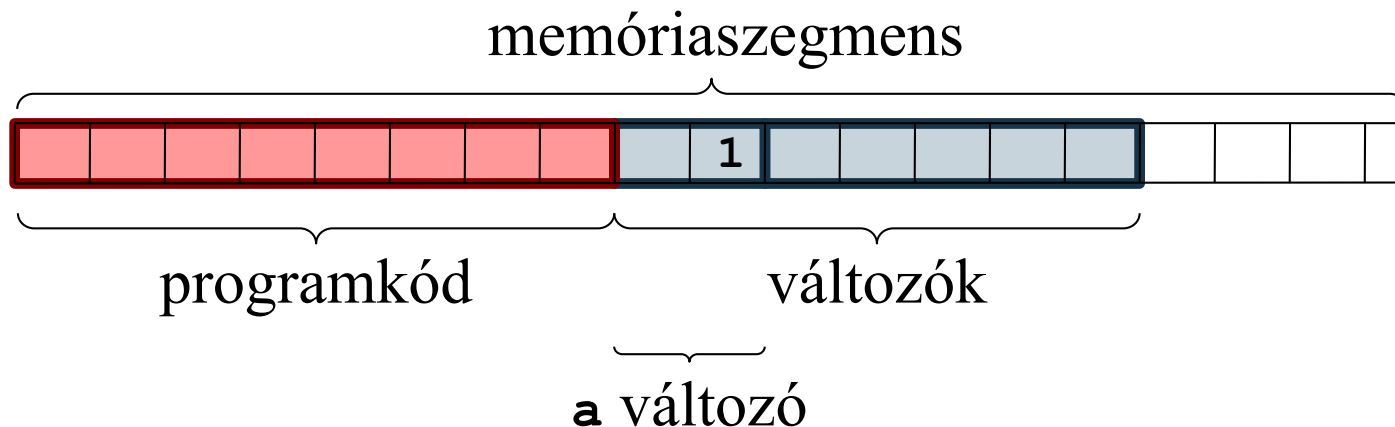


- A adatok kétféle fajtáját tarjuk nyilván:
 - *változó (variable)*: értéke (esetleg típusa is) változtatható a program futása során, mindig a memóriában tároljuk, és megfelelő azonosítóval (*változónév*) látjuk el
 - pl.: $a = 1$ (az a változó értéke megváltozott 1-re)
 - a változókat *deklarálni* (nevét és egyéb tulajdonságait megadni), valamint *definiálni* kell (megadni a hozzájuk tartozó értéket)
 - *konstans (constant)*: értéke és típusa rögzített, nem változhat a program futása során
 - a konstansokat is elláthatjuk azonosítóval (*nevesített konstansok*)

Szoftverfejlesztés

Adatok a memóriában

- A memória tekinthető egy byte-sorozatnak, ahol minden byte-nak sorszáma van, ez a *memóriacím*
 - ehhez rendeljük az azonosítót, amivel a program futása során hivatkozhatunk rá
 - minden futó program önálló memóriaterületet (*szegmenst*) kap, amelyen az ő kódja és az ő adatai helyezkednek el



Szoftverfejlesztés

Adatok típusa

- Az *adattípus* meghatározza a felvehető értékek halmazát (a megvalósításban a memóriában elfoglalt terület mértékét is), valamint az alkalmazható műveletek körét, vagyis

adattípus = értékhalmoz + művelethalmaz

- Minden adat rendelkezik típussal (deklarációkor adjuk meg)
- A gépi kódon értelmezhető, egyszerű típusokat nevezzük *elemi*, vagy *primitív típusoknak*:
 - *logikai* (\mathbb{L})
 - *egész* (\mathbb{Z}), *természetes* (\mathbb{N}) és *valós* (\mathbb{R}) számok
 - *karakter* (\mathbb{Ch})

Szoftverfejlesztés

Adatok típusa

- A programozó alkothat saját, *összetett típusokat* a *típuskonstrukciók* segítségével:
 - *iterált, vagy sorozat* (\mathbb{D}^n , vagy \mathbb{D}^*)
 - *direktszorzat* ($\mathbb{D}_1 \times \mathbb{D}_2$)
 - *unió* ($\mathbb{D}_1 \cup \mathbb{D}_2$)
- A primitív típusokból tetszőleges bonyolultságú típust hozhatunk létre, amelyek művelethalmazát is definiálhatjuk
 - a típuskonstrukciókat halmazozottan is használhatjuk
- Pl.: szöveg ($\mathbb{S} = \mathbb{Ch}^*$), komplex szám ($\mathbb{C} = \mathbb{R} \times \mathbb{R}$), komplex számok vektora ($\mathbb{C}^n = (\mathbb{R} \times \mathbb{R})^n$)

Szoftverfejlesztés

Utasítások

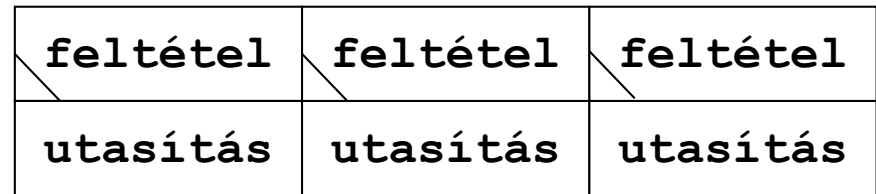
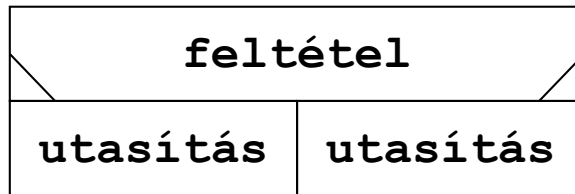
- A programok építőkövei az egyszerű, processzor számára értelmezhető utasítások, az úgynevezett *elemi programok*:
 - *üres program*: **SKIP**
 - *hibás program*: **ABORT**
 - *értékkadás*: **<változó> := <kifejezés>**
- Az elemi programokból különböző *programkonstrukciók* segítségével hozunk létre összetett programokat:
 - *szekvencia*: utasítások egymásutánja

| |
|----------|
| utasítás |
| utasítás |

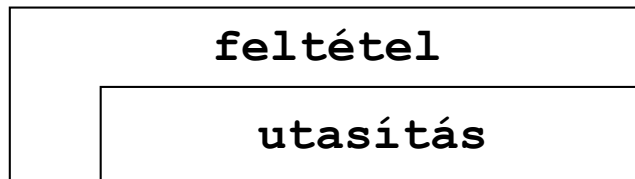
Szoftverfejlesztés

Utasítások

- *elágazás*: feltétel függvényében történő végrehajtás, amely lehet *kétágú*, vagy *többágú*



- *ciklus*: utasítások ismételt végrehajtása, két változata az *előltesztelő* és a *hátultesztelő*



- mindkét esetben a feltétel egy logikai kifejezés

Szoftverfejlesztés

Állapottér, előfeltétel és utófeltétel

- *Állapottérnek* (A) nevezzük a programban előforduló változók összességét (típusukat és elnevezésüket)
- Egy adott futási ponton a változók aktuális értékeinek összességét nevezzük a program *állapotának*
 - a program *állapotot vált*, ha az állapottérben lévő változók közül bármelyik értéke megváltozik
 - a program a *kezdőállapotból* indul, és a *végállapotban* terminál, ezekre külön szabhatunk logikai feltételeket, *előfeltételt* (Q), illetve *utófeltételt* (R)



Szoftverfejlesztés

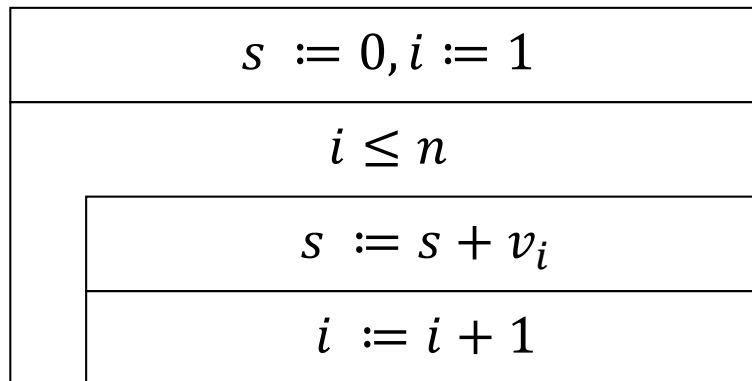
Állapottér, előfeltétel és utófeltétel

- Pl. adjuk össze egy pozitív számokat tartalmazó vektor elemeit:

$$A = (v : \mathbb{Z}^n, s : \mathbb{Z}, i : \mathbb{Z})$$

$$Q = (\forall j \in [1..n]: (v_j = v'_j \wedge v_j \geq 1))$$

$$R = Q \wedge (s = \sum_{j=1}^n v'_j)$$



- *Algoritmus*nak nevezzük azt a műveletsorozatot, amely a feladat megoldásához vezet
 - a program lényegi része, amely nem tartalmazza az adatok beolvasását és kiírását
 - egy programban több algoritmus is szerepelhet, amelyek valamilyen kombinációja oldja meg a feladatot
- A megoldandó feladatok sokszor hasonlítanak egymásra
 - ennek köszönhetően a megoldó algoritmusuk is hasonló, csupán néhány eltérést fedezhetünk fel közöttük
 - sokszor a megfelelő adatok és feltételek cseréjével megkapjuk az új feladat megoldását a korábbi alapján

- Az algoritmusokat ezért célszerű általánosan (absztraktnan) megfogalmazni, hogy a változtatások (transzformációk) könnyen véghezvihetők legyenek
 - ha a feladathoz találunk megoldó algoritmust, akkor azt mondjuk, hogy a feladatot *visszavezettük az algoritmusra*
- Az algoritmust általában két részre szeparáljuk:
 - *inicializálás*: felhasznált változók kezdőértékeinek megadása
 - *feldolgozás* (mag): műveletvégzés a bemenő adatokkal és az inicializált változókkal

- Létező algoritmusokat azért célszerű használni, mert jó, bizonyított megoldását adják a feladatnak
 - már több ezer algoritmus létezik, amelyek mind nevesítettek
- Az egyszerű, sorozatokra alkalmazható algoritmusokat nevezzük *programozási tételek*nek, ezek a következők:
 - összegzés, számlálás
 - lineáris keresés, bináris keresés
 - maximum keresés, feltételes maximumkeresés
 - elemenkénti feldolgozás
- Számos további feladatra adható algoritmus, pl. rendezés, mintaillesztés, adattömörítés, ...

Szoftverfejlesztés

Algoritmusok

- Pl. a lineáris keresés (egy d adatsorozaton keressük az első olyan elemet, amely teljesíti a $\beta: \mathbb{D} \rightarrow \mathbb{L}$ feltételt, jelölje l , hogy sikerült-e megtalálni, és ind a helyét a sorozatban):

$$A = (d : \mathbb{D}^n, ind : \mathbb{Z}, l : \mathbb{L})$$

$$Q = (\forall j \in [1..n]: d_j = d'_j)$$

$$R = Q \wedge \left(\begin{array}{l} l = \exists j \in [1..n]: \beta(d_j) \wedge \\ l \rightarrow (ind \in [1..n] \wedge \beta(d_{ind}) \wedge \forall j \in [1..ind - 1]: \neg \beta(d_j)) \end{array} \right)$$

| |
|-----------------------------|
| $l := \downarrow, i := 1$ |
| $\neg l \wedge i \leq n$ |
| $l := \beta(d_i), ind := i$ |
| $i := i + 1$ |

Szoftverfejlesztés

A programozási nyelv

- Az adat-, illetve utasításabsztrakciót megvalósító eszközt nevezzük *programozási nyelv*nek
 - egy adott programozási nyelven megírt programkódot nevezünk a program *forráskódjának* (*source code*)
 - a programozási nyelv meghatározza az absztrakció szintjét, a használható típusok és utasítások halmazát
 - egy adott nyelvre rögzítettek, ám a legtöbb nyelvben a programozó által kiterjeszthetőek
 - a nyelvet meghatározza a célja, vagyis milyen feladatkörre alkalmazható, továbbá a nyelv rendelkezik egy *kifejezőerővel*, azaz milyen összetett számításokat képes kifejezni

Szoftverfejlesztés

A programozási nyelv

- A programozási nyelvek osztályozása:
 - *alacsony szintű* (assembly): a gépi kódot egyszerűsíti szövegszerűre, de nem biztosít absztrakciót, pl.:

```
data segment ; adatok
```

```
    number dw -5 ; változó létrehozása
```

```
data ends
```

```
code segment ; utasítások
```

```
...
```

```
mov ax, number ; regiszterbe helyezése
```

```
cmp ax, 0 ; regiszterérték összehasonlítása
```

```
jge label1 ; ugrás, amennyiben nem negatív
```

```
mov cx, 0
```

```
sub cx, ax ; pozitívvá alakítás kivonással
```

```
...
```

Szoftverfejlesztés

A programozási nyelv

- *magas szintű*: a gépi architektúrától független utasításkészlettel, nagyobb kifejező erővel rendelkezik
 - lehetőséget ad az utasítás- és adataabsztrakcióra
 - egy egyszerű reprezentációját adja az alacsony szintű kódnak
 - pl.:

```
int main() {  
    int number = -5; // változó létrehozása  
    if (number < 0) // ha negatív  
        number = -number; // ellentettre váltás  
    ...  
}
```

Szoftverfejlesztés

Programkód átalakítás

- A programozási nyelven megírt kódot át kell alakítani a processzor számára értelmezhetővé, ennek módszerei:
 - *fordítás, vagy szerkesztés (compilation)*: a kódot előzetesen lefordítjuk gépi kódra, és ezt az állományt futtatjuk
 - a program gyorsan futtatható, de gépfüggő (pl. nem indul más operációs rendszeren)
 - a műveletet a *fordítóprogram (compiler)* végzi
 - pl.: C, C++, Ada, Pascal, Eiffel, Haskell, Visual Basic
 - *értelmezés (interpretation)*: a programkódot a futtatás közben alakítjuk át gépi kóddá
 - a program gépfüggetlen, de lassú a futtatása

Szoftverfejlesztés

Programkód átalakítás

- ezt a műveletet az *értelmező (interpreter)* végzi, amelynek a futtatás helyén rendelkezésre kell állnia
- pl.: HTML, PHP, JavaScript, Perl, Lisp, MATLAB
- *futásidejű fordítás (just-in-time compilation)*:
 - a fordítás egy része előre megtörténik egy gépfüggetlen, de alacsony szintű, köztes nyelvre (*intermediate language*)
 - a futtatásakor kerül értelmezésre a köztes nyelvű kód, ezt az úgynevezett virtuális gép végzi
 - pl.: Smalltalk, Java, .NET (C#, VB.NET, ...)

Szoftverfejlesztés

Programkód átalakítás

- A programkód átalakítása rendszerint több lépésben történik, magasabb szintű nyelv esetén először alacsonyabb szintű kód, majd abból gépi kód készül
- A programkód tartalmazhat hibákat, amelyeket két kategóriába sorolunk:
 - *szintaktikai, vagy elemzési hibák (syntax error)*: a programkód szerkezete helytelen, pl. hibás utasításnév, hivatkozás, zárójelezés, ...
 - *szemantikai, vagy értelmezési hibák (semantic error)*: az érték változásával, a műveletek végrehajtásával bekövetkező hibák, pl. 0-val történő osztás, hibás memóriacím hivatkozás, ...

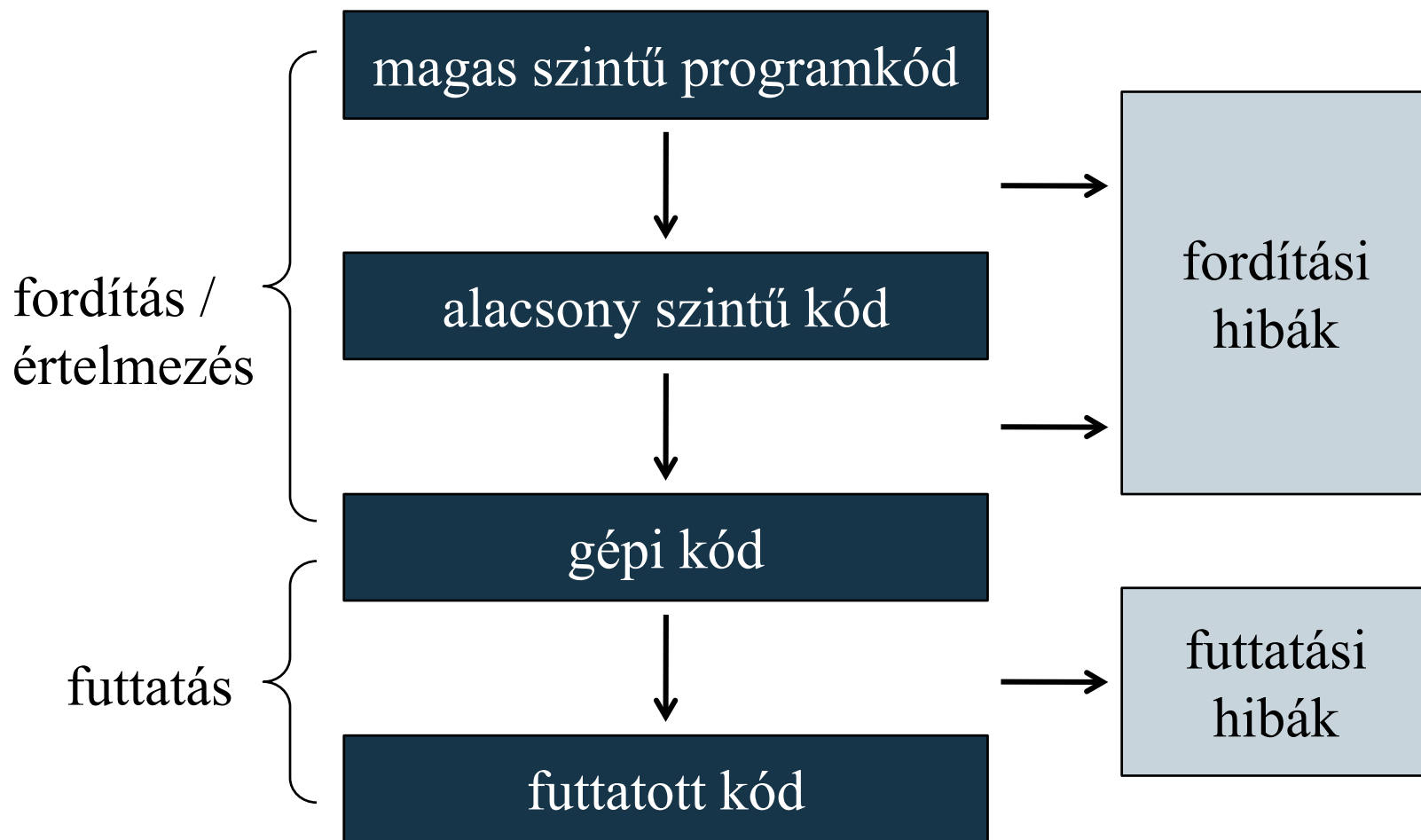
Szoftverfejlesztés

Programkód átalakítás

- A programhibákról az átalakítás során a lehető legpontosabb információt kapjuk (lehetséges ok és hely megadásával)
 - a fordítás során a szintaktikus hibák előre kiszűrhetőek, a szemantikus hibák nagy része azonban futtatáskor léphet fel
 - értelmezés esetén csak futtatáskor szembesülünk a hibákkal
- A további programhibák ellenőrzését *teszteléssel* végezhetjük
 - a *statikus tesztelés* során a programkódot vizsgáljuk át
 - a *dinamikus tesztelés* során futás közben keressük a hibákat
- A programfejlesztői környezetek megadják a *nyomkövetés* (*debug*) lehetőségét is (futás közben végigkövethetjük a kódot)

Szoftverfejlesztés

Programkód transzformáció



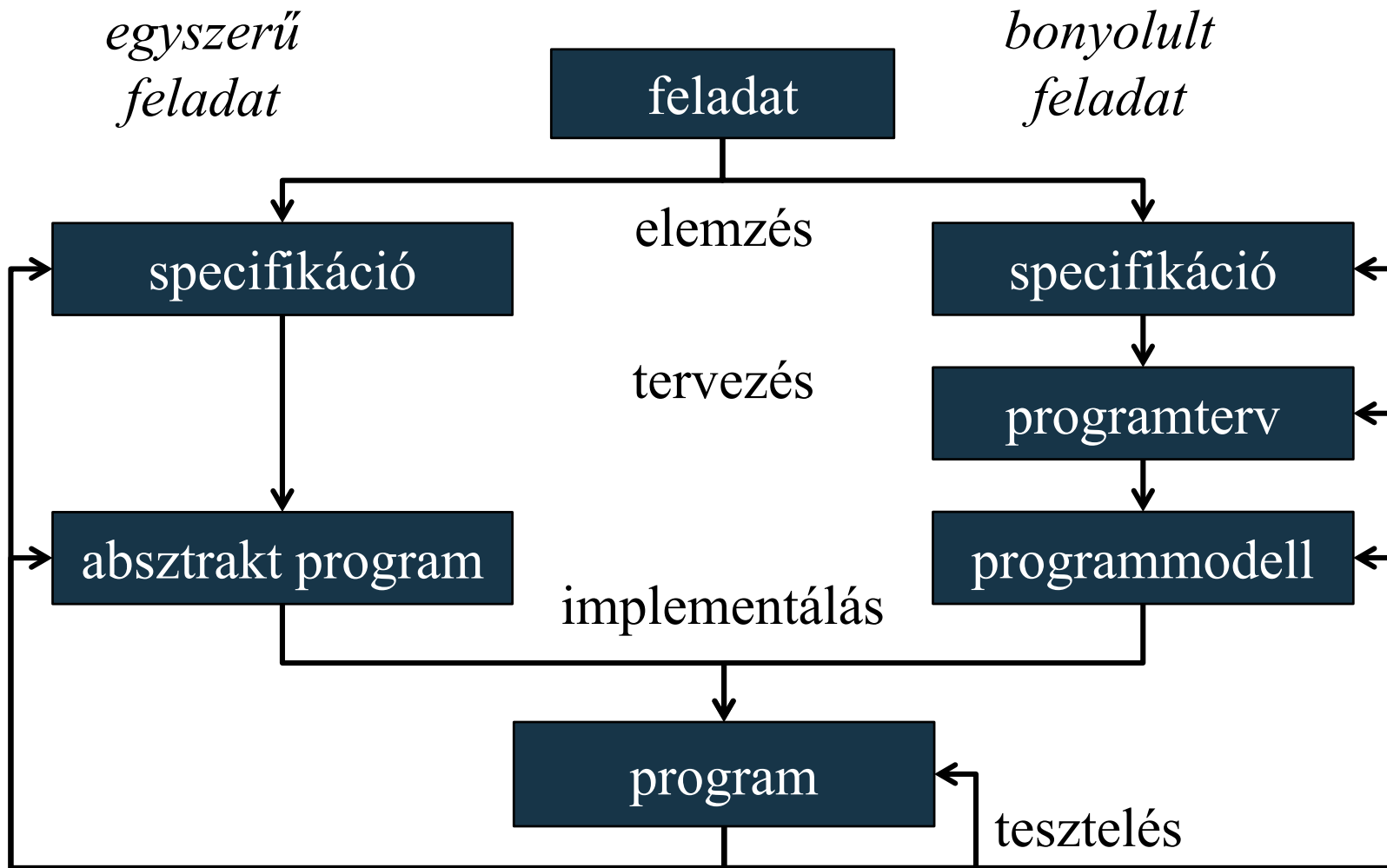
Szoftverfejlesztés

A szoftverfejlesztés folyamata

- A szoftverfejlesztés a kódoláson túl több lépésből áll, amely függ a feladat bonyolultságától is:
 1. A feladatot elemezni kell, és megadni a formális megfelelőjét, vagyis a *specifikációt*
 2. A specifikációt alapján megtervezhető a program, amely egyszerű feladatnál az *absztrakt program*, míg bonyolult feladatnál a *programterv* elkészítésével jár, amelyből elállítható a *programmodell* (egyszerűsített célprogram)
 3. A tervet implementáljuk a megfelelő programozási nyelven
 4. Az implementált programot, illetve a programkódot *tesztelésnek* vetjük alá, ami módosításokat eredményezhet az implementációban (vagy a korábbi fázisokban)

Szoftverfejlesztés

A szoftverfejlesztés folyamata



Szoftverfejlesztés

A feladat elemzése és tervezése

- Önmagában a feladat elemzése is nagyban meghatározza a programfejlesztés folyamatát, ennek két formája:
 - *felülről lefelé (top-down)*: a főfeladatot részfeladatokra, majd azokat további részfeladatokra bontjuk
 - *alulról felfelé (bottom-up)*: a feladatban szereplő egységeket határozzuk meg, majd azokat kombináljuk
- A tervezés során egy nyelv-független vázát kell elkészítenünk a szoftvernek, amely megadja annak közeli működését
 - lehet formális, vagy informális modell
 - használhatunk modellező eszközt (pl. stuktogram), vagy nyelvet (pl. UML)

Szoftverfejlesztés

Tesztelés

- A *tesztelés* annak ellenőrzése, hogy a program teljesíti-e az előírt funkcionalitást, illetve minőségi elvárásokat
 - célja elsősorban a futási idejű hibák, működési rendellenességek keresése
 - a tesztelés módja szerint lehet:
 - *fekete doboz tesztelés*: a tesztelendő programrész ismeretlen, csak a hiba voltát fedezzük fel
 - *fehér doboz tesztelés*: a programrész teljes mértékben ismert, tehát így a hiba helyét is megtalálhatjuk
 - a tesztelés módszere lehet *statikus* (kód kiértékelés és ellenőrzés), illetve *dinamikus* (futtatás adott tesztesetekkel)

Szoftverfejlesztés

A szoftverfejlesztés optimalizálása

- A szoftverfejlesztés során a legfőbb cél, hogy a kész program megfeleljen a *funkcionális* és *minőségi követelményeknek*
 - emellett, a fejlesztők számára fontos, hogy a kész szoftver fejlesztése a lehető legoptimálisabb legyen
- A szoftverek tervezésének és programozásának módszerét nevezzük *programozási paradigmának*
 - meghatározza a programozási stílust, az absztrakciós szintet
 - meghatározza az alkalmazható programozási nyelvek körét is, és fordítva
 - sok programozási nyelv több paradigmát is támogatnak, ezek a *multiparadigma* nyelvek

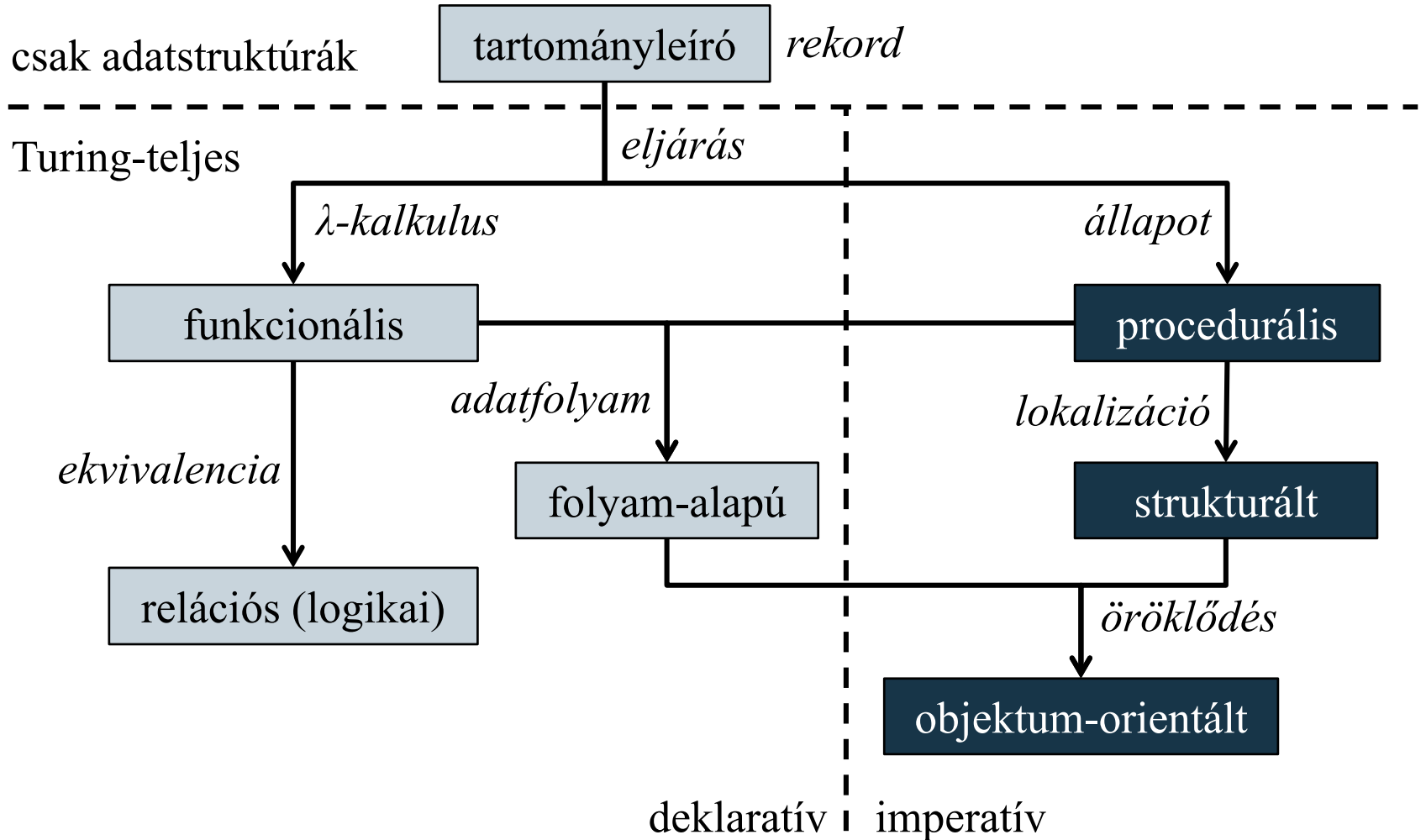
Programozási paradigmák

A paradigmák csoportosítása

- A programozási paradigmákat két csoportba soroljuk:
 - *imperatív*: a programot állapotváltozások sorozatával írja le
 - az utasításokat szekvenciálisan hajtja végre
 - megfelel a gépi szintű végrehajtásnak
 - *deklaratív*: a program a tartalmát, megjelenését írja le, nem pedig a funkció végrehajtásának módját
 - nem alkalmaz változókat, csak konstans értékeket
 - az utasításokat nem feltétlenül szekvenciálisan hajtja végre, automatikusan párhuzamosít
 - magasabb szintű kifejezőerővel rendelkezik

Programozási paradigmák

A jelentősebb paradigmák



Programozási paradigmák

Procedurális programozás

- *Procedurális (Procedural)*:
 - a programot *alprogramokra (subroutine)* bontja, és minden alprogram meghatározott részfeladatot végez el
 - az alprogramoknak két típusa lehet:
 - *eljárás (procedure)*: valamilyen utasítássorozatot futtat, végeredmény nélkül
 - *függvény (function)*: valamilyen matematikai számítást végez el, és megadja annak eredményét
 - az alprogramok programkonstrukciókkal épülnek fel, meghívhatnak más alprogramokat, és kommunikálhatnak velük

Programozási paradigmák

Procedurális programozás

- a vezérlést a főprogram szolgáltatja, amely kezeli a teljes programban jelen lévő adatokat
- nyelvek: *Fortran*, *C*, *BASIC*, *Pascal*
- pl. (C++, vektor összegzése függvénnyel):

```
int Sum(vector<int> values) {  
    // a függvény paraméterben megkapja a vektort  
    int sum = 0;  
    for (int i = 0; i < value.size(); i++)  
        sum = sum + values[i];  
    // ciklussal hozzávesszük ez elemeket  
    return sum; // visszatérési érték az összeg  
}
```

Programozási paradigmák

Procedurális programozás

- pl. (Fortran, tömb átlaga):

```
implicit none
```

```
integer :: number_of_points real,
```

```
dimension(:), allocatable :: points
```

```
real :: average_points=0.
```

```
read (*,*) number_of_points
```

```
allocate (points(number_of_points))
```

```
read (*,*) points
```

```
if (number_of_points > 0)
```

```
    average_points = sum(points)/number_of_points
```

```
deallocate (points)
```

```
write (*, '( 'Average = ' , 1g12.4) ')
```

```
    average_points
```

Programozási paradigmák

Procedurális programozás

- pl. (Maple, szám faktoriálisa):

```
myfac := proc(n::nonnegint)
    local out, i;
    out := 1;
    for i from 2 to n do
        out := out * i
    end do;
    out
end proc;
```

ugyanez lambda-kifejezés segítségével (funkcionálisan):

```
myfac := n -> product( i, i = 1..n );
```

Programozási paradigmák

Strukturált programozás

- *Strukturált (Structured)*:
 - a program részegységekre (*csomagokra*, vagy *blokkokra*) tagolódik, minden egység rendelkezik belépési ponttal, és kilépési ponttal
 - a programegységeknek van egy kívülről látható része (*interfész*), és egy belső megvalósítása (*implementáció*)
 - a programban használt adatstruktúrák a programegységeknek megfelelően strukturálódnak
 - támogatja a kivételkezelést, tiltja a programkódban történő ugrást (*goto*)
 - nyelvek: *Pascal*, *C*, *ADA*

Programozási paradigmák

Strukturált programozás

- pl. (Ada, verem csomag):

```
package STACK_T is
    type Stack is limited private;
    procedure Push (v: in out Stack; e: Value);
    procedure Pop (v: in out Stack);
    function Top (v: Stack) return Value;
private
    type Node;
    type Pointer is access Node;
    type Node is record
        data: Value; next: Pointer := null;
    end record;
    ...
end STACK_T;
```

Programozási paradigmák

Strukturált programozás

- pl. (C++, verem típus):

```
class Stack {  
private: // rejtett rész, implementáció  
    int* values; // attribútumok  
    int top;  
public: // látható rész, interfész  
    Stack() { values = new int[10]; top = 0; }  
    // konstruktor  
    ~Stack() { delete[] values; } // destruktork  
    void Push(int v) { // metódus  
        if (top < 10) { values[top] = v; top++; }  
    }  
    ...  
};
```

Programozási paradigmák

Objektum-orientált programozás

- *Objektum-orientált (Object-oriented)*:
 - a feladat megoldásában az alulról-felfelé megközelítést alkalmazza, alapja az *egységbe zárás* és az *öröklődés*
 - a programot egymással kommunikáló objektumok alkotják, amelyek valamilyen relációban állnak egymással
 - manapság a legnépszerűbb programozási paradigma, a programozási nyelvek jelentős része támogatja
 - objektumorientált támogatással rendelkező nyelvek: *C++*, *Objective-C*, *Matlab*, *PHP*, *Python*, *Perl*, ...
 - tisztán objektumorientált nyelvek: *Smalltalk*, *Java*, *C#*, *Eiffel*, *Ruby*, ...

Programozási paradigmák

Objektum-orientált programozás

- pl. (C++, grafikus felületű ablak):

```
class DemoWindow : public QWidget {
    // ablak osztály
public:
    DemoWindow(QWidget* parent = 0) {
        // a konstruktor megkaphatja a szülőt
        setBaseSize(200, 120);
        setWindowTitle("Demo Window");
        QPushButton = new QPushButton("Quit", this);
        // gomb példányosítása
    }
private:
    QPushButton* QPushButton; // gomb az ablakon
};
```

Programozási paradigmák

Objektum-orientált programozás

- pl. (Java, mobil alkalmazás vezérlője):

```
public class HelloWorld extends MIDlet
    // a MIDlet osztály leszármazottja
    implements CommandListener {
    private Form form; // implementáció
    private Command exitCommand;

    public HelloWorld() { // konstruktor
        form = new Form("Welcome!", new Item[]
            { new StringItem("", "Hello World!") });
        form.addCommand(exitCommand);
        form.setCommandListener(this);
    }
}
```