



# Programozási technológia I.

## *Objektumelvű technológia*

Objektumorientált  
tervezés

Az UML használata

Példa

Naív modell

A naív modell problémái

Javított modell

Dr. Szendrei Rudolf  
Informatikai Kar  
Eötvös Loránd Tudományegyetem



Objektumorientált  
tervezés

Az UML használata

Példa

Naív modell

A naív modell problémái

Javított modell

# Tartalom

## 1 Objektumorientált tervezés

## 2 Az UML használata

## 3 Példa

Naív modell

A naív modell problémái

Javított modell



# Objektumorientált tervezés

## Az objektumorientált programozás kialakulása

### Programozás fejlődése:

- gépi kód
- mnemonikok (assembly)
- vezérlési szerkezetek, eljárások (FORTRAN)
- adattípusok (Pascal, C)
- típusosztályok (Ada)
- objektumosztályok (C++, Java)

konkrét

absztrakt



# Objektumorientált tervezés

## Procedurális tervezés

- Funkciókból indul ki, azokat dekomponálva alakul ki a szerkezet.
- Közös erőforrást (adatszerkezetet) használó műveletek csoportosítása.
- A modulokhoz szabványos hozzáférési felület tartozik. (Típus)
- Típusöröklődés.



# Objektumorientált tervezés

## Objektumelvű tervezés

- Nem a funkciókból, tevékenységekből indulunk ki, hanem az adatokból, a feladatban részt vevő elemekből.
- Ezeket azonosítjuk, csoportosítjuk, felderítjük kapcsolataikat, felelősségeiket. Így jönnek létre objektumok, illetve osztályok.
- A rendszer funkcionalitását az egymással együttműködő objektumok összessége adja ki. Egy objektum csak egy jól meghatározott részért felelős.
- Az objektumok adatot tárolnak, ezek kezeléséért felelősek, de ezeket elrejtetik a külvilág elől. Szabványos módon lehet az objektumokkal kapcsolatba lépni.



# Objektumorientált tervezés

## Eszközök

- Objektumorientált tervezést támogató nyelvek: C++, Objective-C
- Tisztán objektumorientált nyelvek: Java, C#
- Szabványos tervezési nyelv (UML)
- Ezekre épülő integrált fejlesztői eszközök



# Az UML használata

## Tervezés

- Az UML segítségével szabványos módon lehet szoftverek terveit elkészíteni
  - alkalmas üzleti folyamatok és programfunkciók, és adatbázis-sémák leírására
  - a modellek automatikusan kódba fejthetőek, tehát tetszőleges objektumorientált nyelvre átírhatóak
  - a nyelv kiterjeszthető, és lehetőséget ad a személyesítésre
  - a nyelv leginkább diagramok alakjában jelenik meg, noha a nyelv nem a diagramokat magukat adja meg, hanem a diagramok által reprezentált modell specifikációját
- 2003-ban készült el az UML 2.0-s specifikációja, amely 13 diagramtípust vezet be.



# Az UML használata

## A rendszer jellemzései az UML segítségével

A szoftver fejlesztésének életciklusát folyamatosan követi a modellezés. Az UML e tekintetben egy öt pillérű szemléletrendszerrel rendelkezik:

- *Használati*: a szoftver szolgáltatásai és azok kapcsolata a felhasználókkal
- *Szerkezeti* (statikus): a rendszer és programegységek felépítése, kapcsolatai
- *Dinamikus*: a programegységek viselkedése
- *Implementációs*: a megvalósítás szempontjai, komponensei
- *Környezeti*: hardver és szoftver erőforrások

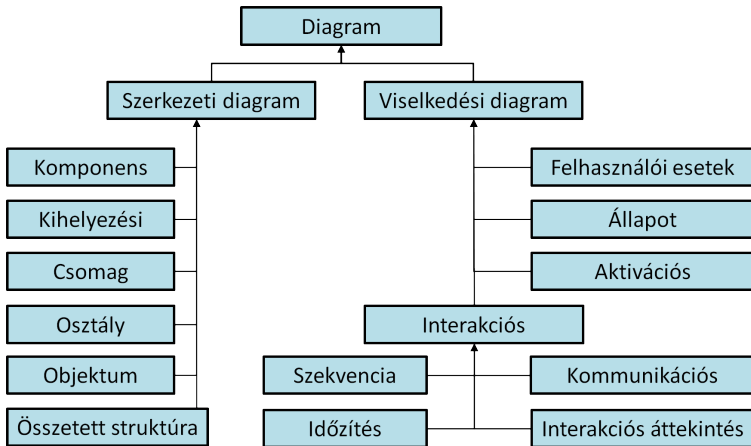




# Az UML használata

## Az UML diagramjai

- A szoftverfejlesztés különböző szakaszaiban az UML különböző diagramjait kell alkalmaznunk:
  - elemzés: felhasználói esetek, komponens, kihelyezési
  - specifikáció: komponens, telepítési
  - tervezés:
    - statikus tervezés: csomag, osztály, objektum
    - dinamikus tervezés: állapot, szekvencia, tevékenység, interakciós áttekintési, kommunikációs
  - tesztelés: időzítés
- A későbbi fázisokban a korábban létrehozott diagramok újra alkalmazhatóak, esetleg módosíthatóak
- A diagramoknak a fázist követő dokumentációban szerepelniük kell





## Példa

### LEGO építmények gyári kezelése

- Elemekből rakhatunk össze építményeket
- Egy építmény tartalmazhat elemeket és részépítményeket vegyesen
- Részépítmények tetszőleges mértékben egymásba ágyazhatóak

### Feladat

Határozzuk meg egy építmény árát, ami a benne szereplő elemek árainak összessége.

### További feladatok

- Egy építmény elemeinek kilistázása,
- egyes elemekből mennyi felhasználható elem van készleten.



## Példa - LEGO építmények gyári kezelése

### Elemek

A példa szempontjából a következőket kell ismernünk egy elemről:

- név (string)
- szín
- ár (egész)



## Példa - LEGO építmények gyári kezelése

### Építmények

- Az elemek összeszerelhetők bonyolultabb szerkezetekbe, amelyeket építményeknek nevezünk.
- Egy építmény tetszőleges számú elemből állhat, és hierarchikus szerkezete lehet, vagyis tartalmazhat építményeket is.
- A tartalmazott építményt a továbbiakban részépítménynek nevezzük.
- Egy részépítményben lehetnek elemek, illetve további részépítmények.



## Példa - LEGO építmények gyári kezelése

### Objektumok

- Első lépésben a rendszer objektumait azonosítjuk.
- A tervezés során az egyik legnehezebb feladat a rendszer adatainak felosztása objektumok halmazába úgy, hogy az objektumok sikeresen működjenek együtt a rendszer teljes működésének megvalósításában.
- Egy gyakori ökölszabály az objektumok kiválasztására, hogy a valóságos elemeknek a modellben objektum feleljen meg.
- Rendszerünk egyik fő feladata, hogy nyomon kövesse az összes elemet.
- Ezért adódik, hogy minden elemet objektumként kezeljünk a rendszerben.



## Példa - LEGO építmények gyári kezelése

### Objektumok

- Sokféle elemobjektum fordulhat elő, amelyek különböző elemeket írnak le, de mindegyiknek ugyanaz lesz a szerkezete.
- Ugyanazt a valóságelemet kifejező objektumhalmaz közös szerkezetét osztállyal írjuk le.
- Az objektumhalmaz minden eleme egy példánya lesz az osztálynak.
- Az osztály egyrészt tartalmazza a közös szerkezetet (adatok), másrészt az objektumokon végrehajtandó műveleteket.
- Esetünkben az `Elem` osztály létrehozása a tervezés első lépése.



## Példa - LEGO építmények gyári kezelése

### Elem osztály (Java, UML)

```
import java.awt.Color;

public class Elem
{
    private String név;
    private Color  szín;
    private int    ár;

    public Elem(String név,
                  Color  szín,
                  int    ár)
    {
        this.név  = név;
        this.szín = szín;
        this.ár   = ár;
    }

    public String név() { return név; }
    public Color  szín() { return szín; }
    public int    ár()  { return ár; }
}
```

Elem
-név: String -szín: Color -ár: int
+Elem(név:String, szín:Color, ár:int) +név(): String +szín(): Color +ár(): int





## Példa - LEGO építmények gyári kezelése

### Elem osztály

Az osztályok fordítási időben lesznek meghatározva, az objektumok viszont futási időben jönnek létre, mint az osztályok példányai.

```
Elem e = new Elem("2x2_kocka", Color.RED, 2);
```

művelet végrehajtása után egy új objektum jön létre.

A memóriában egy terület tartozik az objektumhoz, amely a megfelelő értékekkel rendelkezik.

e : Elem
név = "2x2 kocka"
szín = Color.RED
ár = 2



## Példa - LEGO építmények gyári kezelése

### Azonosíthatóság

- Lényeges eleme az objektum definíciójának, hogy az objektumok megkülönböztethetők egymástól, azaz bármely objektum megkülönböztethető bármely más objektumtól.
- Ez akkor is teljesül, ha két objektum pontosan ugyanazokat az adatokat tartalmazza és felületük is megegyezik.
- Például a következő programrészlet eredménye két objektum, amelyek állapota megegyezik, az objektumok mégis megkülönböztethetők.

```
Elem e1 = new Elem("2x2_kocka", Color.RED, 2);  
Elem e2 = new Elem("2x2_kocka", Color.RED, 2);
```



## Példa - LEGO építmények gyári kezelése

### Azonosíthatóság (folyt.)

- Az objektumelvű modell feltételezi, hogy minden objektumhoz tartozik egy „azonosság”, amely egyfajta címkeként megkülönbözteti az objektumot másoktól.
- Ez az azonosság egy belső, lényeges része az objektumelvű modellnek, és különbözik az objektumban tárolt adatok mindegyikétől.
- Objektumelvű nyelvek esetén az objektum memóriabeli címe használható erre a célra. Ez nyilvánvalóan eltérő különböző objektumok esetén.
- Java: objektumok egyenlőség vizsgálata azok memóriacímét veti össze.
- `this` az objektum saját memóriacíme.



## Példa - LEGO építmények gyári kezelése

### Azonosíthatóság (folyt.)

- UML-ben az objektumokhoz neveket rendelhetünk az osztálynév mellett, és így biztosíthatjuk az objektum egyediségét.
- Ezeket a neveket a modellen belül használhatjuk, és lehetőséget adnak, hogy egy objektumra egyedileg hivatkozzunk a modellben.
- Az objektumnév nem felel meg semmilyen adategységnek.
- Az objektumnév különbözhet annak a változónak a nevétől, amellyel a programban hivatkozunk az objektumra (pl. referenciák).
- Gyakran kényelmes és praktikus, ha a két név megegyezik.
- A programban több változó is hivatkozhat ugyanarra az objektumra, illetve egy változó az élettartama során több objektumra is hivatkozhat. Ezért a névegyezés nem mindig valósítható meg.



Objektumorientált  
tervezés

Az UML használata

Példa

Naív modell

A naív modell problémái

Javított modell

# A naív modell problémái

## Adatismétlés

- A bevezetett modell ugyan kézenfekvő, azonban egy elemtípust leíró adatokat megismételjük minden egyes elem esetén.
- Az ismétlés oka: a leírásokat az objektumokban tároljuk.
- Ha a rendszerben kettő vagy több elem van ugyanabból a fajtából, akkor annyi objektum jön létre, és mindegyik tartalmazza ugyanazokat az adatokat.



# A naív modell problémái

## Az adatismétlés következményei

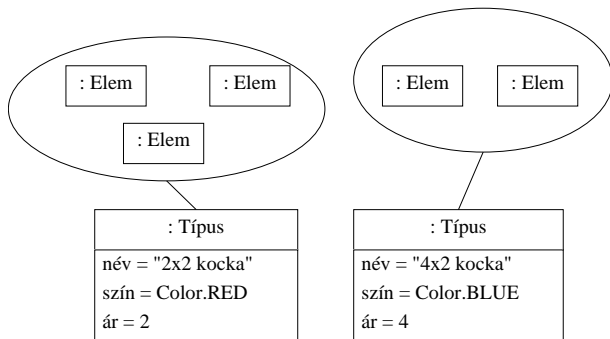
- Jelentős redundanciát eredményez.
- Az ár ismétlése várhatóan karbantartási problémához vezet.
- Egy elemre vonatkozó információt tartósan kell tárolni, most objektumhoz kötött.



## Javított modell

### Az adatismétlés elkerülése

- Az azonos típusú elemeket leíró közös információt egy elkülönített objektumban tároljuk.
- Ezek a „leíró” objektumok nem képviselnek egyedi elemeket, hanem egy csatolt információt tartalmaznak, amely megadja egy elem típusát.
- Nevezzük ezeket az objektumokat *típusnak*.
- Minden egyes elemtípushoz tartozik egy egyedi típus objektum, amely tárolja a nevét, a szint és az árat.
- Az elemeket reprezentáló objektumokban az adatok nem jelennek meg. Ezeket az adatokat a megfelelő típustól kérhetjük le, ezért minden elemnek ismernie kell a megfelelő típust, hivatkoznia kell arra.







# Javított modell

## A javítás elemzése

Az előző problémák mindegyikét megoldjuk így, mert:

- Az adatokat csak egy helyen tartjuk nyilván, így megszűnik a redundancia.
- Egy adott elem adatainak változtatása egyszerű, csak egy típus adatait kell módosítani.
- A típus mindig létezhet, függetlenül attól, hogy mennyi elem objektum található a rendszerben. Így az információ tárolható még az objektum létrejötte előtt.