

Szoftvertechnikák

Futtatókörnyezetek
Benedek Zoltán



Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

Futtatókörnyezetek
általánosságában

JAVA

> JVM

.NET

> Áttekintő

> CLR

> Fordítás folyamata

> Szerelvények

Futtatókörnyezet

- A virtual machine (VM) egy absztrakt számítógép architektúra
- Szoftver egy valódi hardver fölött
- Ugyanaz az alkalmazás futhat különböző operációs rendszerekben, platformokon
- Felügyelheti a kódot és az adatot
- „Wrapper”-ként is lehet értelmezni az OS API fölött
- Alapvetően 2 konkrét esetet nézünk meg
 - > JVM (Java)
 - > CLR (.NET)

Futtatókörnyezetek jellemzői I.

VM, IL, LIEP (Language Independent Execution Platforms) Az utóbbi években ezek a témák kerültek előtérbe, a következők miatt:

- Hordozhatóság:
 - > IL segítségével $n+m$ (n^*m helyett) translator-ra van szükség, hogy n nyelvet, m platformra implementáljunk
- Kompaktság:
 - > Az IL kód kompaktabb is lehet, mint az eredeti kód

Futtatókörnyezetek jellemzői II.

- Hatékonyság:
 - > Köztes kódról natív kódra csak a lehető legkésőbbi pillanatban fordul, a végleges környezetben. Így kihasználhatja annak sajátosságait, valamint statisztikát készíthet a futtatókörnyezet a futó kódról, és erre építve optimalizált kódot fordíthat (pl. első néhány futáskor még interpretáltan futtat, míg statisztikát készít, és csak utána fordít natív kódra).
 - > A fejlesztők egy környezetben dolgoznak, fejlesztésük eredménye több platformon is futtatható (nem kell az egyes platformok sajátosságait megismerni).

Futtatókörnyezetek jellemzői III.

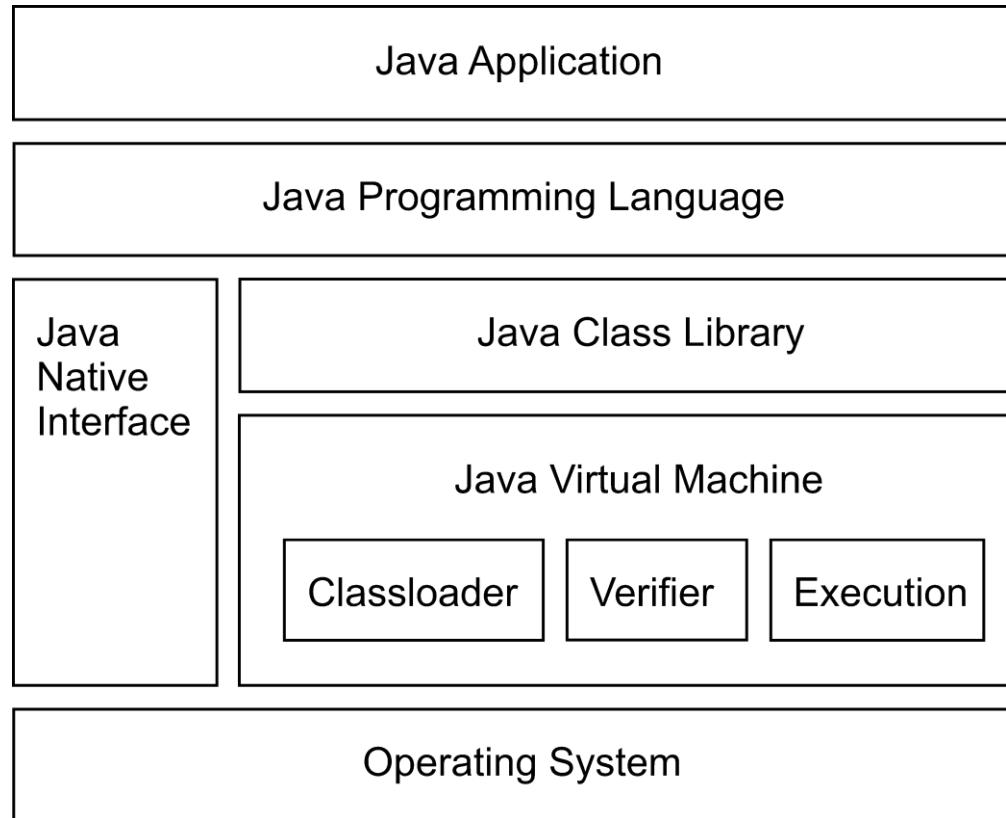
- Biztonság:
 - > Adatellenőrzés (pl. szemétgyűjtő, nincs memóriaszivárgás, stb.)
 - > Kódellenőrzés (pl. futás közbeni hibák azonnali megfogása)
 - A legrosszabb, ha nem derülnek ki azonnal a hibák futás közben, hanem rejtve maradnak jó ideig (pl. C++)
- Együttműködés:
 - > Többnyelvűség könnyebb támogatása (pl. .NET, Java)
- Rugalmasság:
 - > „metaprograming”, reflection, dynamic code generation, serialization, type browsing, etc.
 - Ezek később kerülnek tárgyalásra a félév során.

JAVA

Java tulajdonságok

- Egyszerű és objektum-orientált
 - > Egyszerűsített objektum modell egy egyszeri öröklődéssel
 - > A runtime sorsa Oracle kezében van ?
- Portolhatóság
 - > Java compiler **bytecode**-ot generál
 - > Runtime különböző platformokra
 - > Az alap adattípusok mérete és viselkedése definiált
 - > „*Write once, run/debug anywhere*”

Java



Java tulajdonságok 2

- Elérhetőség
 - > Windows, Linux, stb.
 - > Beágyazott rendszerek
 - > Compiler és runtime ingyenes
 - > Ingyen IDE-k: Eclipse, IntelliJ IDEA
- Library
 - > Gazdag class library
 - > Mobil, Standard, Enterprise
 - > Szabványos GUI
 - > Stb..

Java tulajdonságok 3

- „Built-in model” a konkurencia kezeléshez
 - > Szálkezelés a nyelv része
 - > Szinkronizáció támogatás
- Biztonság
 - > **Nincs pointer!**
 - > Compile-time ellenőrzés
 - > Runtime ellenőrzés
 - > **Automatikus memória menedzsment - GC (Garbage Collector)**

JVM

- Futtatási környezet Java-hoz
- A működése szabványban rögzített (bytecode utasításkészlet, stb.)
- Több implementáció is létezik
- Futtat Java .class fájlokat
- Meg kell felelnie az Oracle (régen Sun) specifikációnak

JVM implementációi

- Interpreter
 - > Egyszerű, kompakt
 - > Lassú
- Just-in-time compilation (JIT)
 - > Implementációfüggő
 - > Egy modern megoldás: első N alkalommal interpretál, statisztikát készít, majd lefordítja a bytecode-ot. Ettől kezdve gyorsabb a végrehajtás.
- Hardver implementáció
 - > Java bytecode-nak megfelelő CPU utasításkészlet
 - > Egyelőre nem nagyon terjedt el

.NET

.NET Fogalmak

- .NET Framework (keretrendszer)
 - > Common Language Runtime - (CLR)
 - Közös nyelvi futtatókörnyezet
 - GC, JIT fordító stb.
 - > Base Class Library - BCL
 - Alap osztálykönyvtárak (string, file, gyűjtemények, szálak, stb. kezelésére)
 - > Számos beépített szolgáltatás (lásd később)
 - > NuGet csomagok formájában számtalan kiegészítő könyvtár letölthető

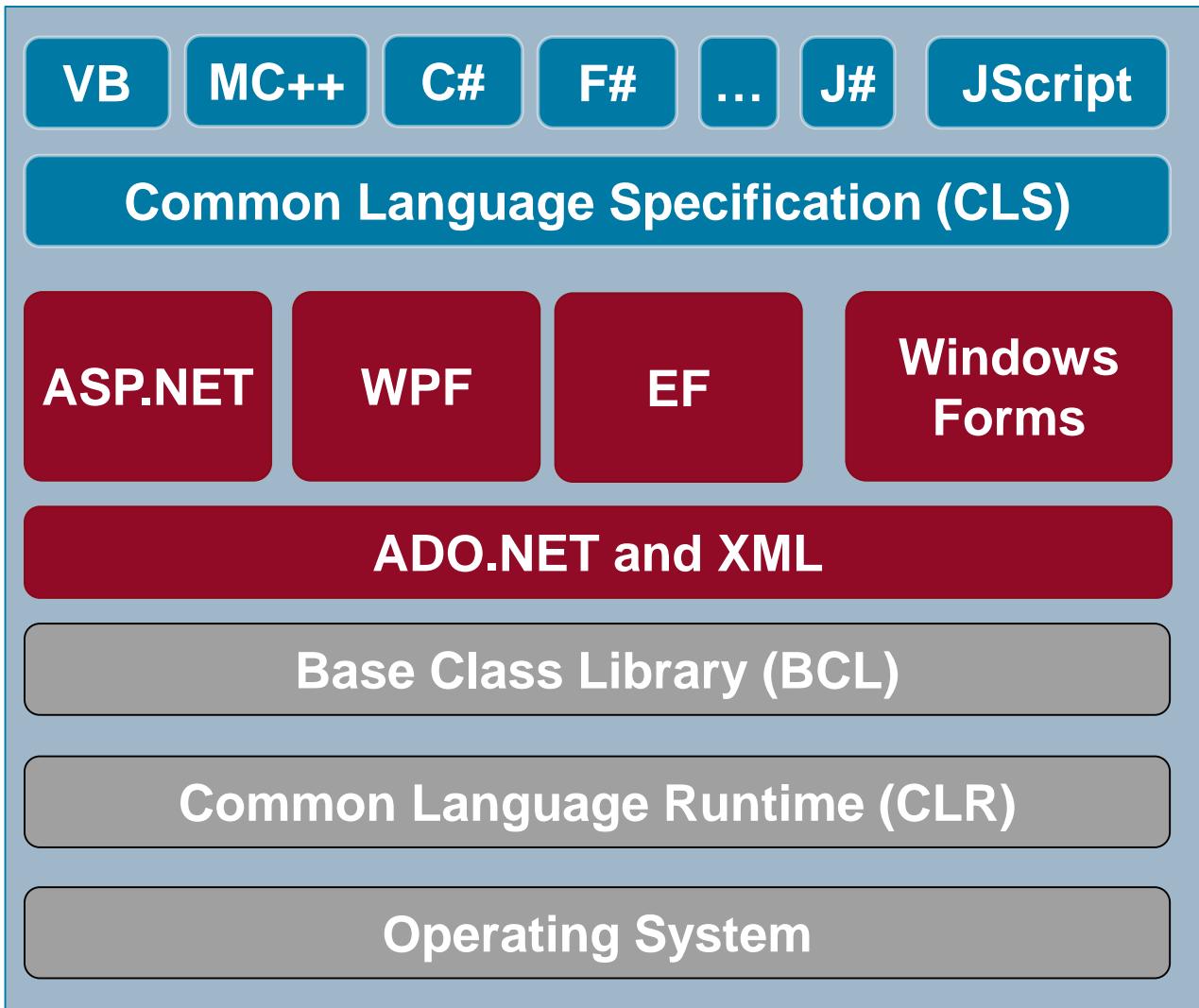
.NET technológiák

- Konzol alkalmazások (lesz a tárgy keretében)
- Webes frontend alkalmazások
 - > ASP.NET/ ASP.NET Core
- Backend szolgáltatások
 - > Web API (REST)
 - > Windows Communication Foundation (WCF)
- Desktop alkalmazások
 - > Windows Forms (lesz a tárgy keretében)
 - > Windows Presentation Foundation (WPF)
 - > Universal Windows Platform (UWP)
- Vastag-, mobilkliens alkalmazások
 - > Xamarin (Mono-t használ)
- Játékfejlesztés
 - > Unity
- Adatkezelés
 - > ADO.NET (lesz a tárgy keretében)
 - > Entity Framework
- Stb.

.NET (runtime) változatok

- **.NET Framework** (elsődlegesen ez lesz a tárgyból)
 - > Sokáig csak ez létezett
 - > Csak Windows platform
 - > Legtöbb szolgáltatást/funkciót ez nyújtja
- **.NET Core**
 - > Keresztplatformos implementáció
 - Windows, Linux, Mac
 - > Szűkebb funkcionális – de folyamatosan bővül
 - A Microsoft ezt fejleszti jelenleg aktívan
- **Mono**
 - > Keresztplatformos implementáció
 - Windows, Linux, Mac, Android, iOS (Xamarin), Unity3D
 - > Funkciókban valahol a Full és a Core között
- **.NET Standard**
 - > Ez nem implementáció, hanem **szabvány** a változatok közötti forráskód hordozhatóság érdekében
 - Full .NET, .NET Core, Mono ennek az implementációi
 - > Több verzió

Keretrendszer, nyelvek, eszközök



.NET

CLR

CLR – Common Language Runtime

- Közös nyelvi futtatókörnyezet
- Tervezési szempontok
 - > Egyszerűbb alkalmazásfejlesztés
 - > Több programnyelv támogatása/integrációja
 - Common Language Specification - CLS
 - > Típus kompatibilitás
 - Common Type System - CTS
 - > Robosztus és biztonságos **felügyelt (managed)** futtató környezet
 - IL, GC, CAS, JIT ...
 - > Egyszerűbb telepítés és üzemeltetés

Egyszerűbb fejlesztés I.

- Objektumorientált környezet
 - > A futtatórendszer típusrendszere objektumorientált
 - > Egyszeres öröklés
 - > Tetszőleges számú interfész implementálható
- Keretrendszer beépített osztályai
 - > Hierarchikus
 - Osztályokba és névterekbe szervezett kód
 - > Örökölhetők, testreszabhatók
- Egységes, gazdag típusrendszer
 - > minden osztály

Egyszerűbb fejlesztés II.

- Komponens alapú fejlesztés támogatása
 - > A tulajdonságok, események, attribútumok nyelvi szinten jelennek meg
 - > Gazdag design-time funkcionálitás a fejlesztőeszközben
- Zökkenőmentes integráció korábbi technológiákkal
 - > Win32 API elérhető: PInvoke (Platform Invoke)
 - > COM, COM+ Enterprise Services, C++ natív kód, stb..
 - > COM (és ActiveX)
 - COM komponensek felhasználhatók
 - .NET objektumok COM komponensként „láttathatók”

CLR komponensek

Osztálykönyvtár-kiszolgálás

Szálkezelő

COM Marshaler

Típusellenőrző

Kivételkezelő

Biztonsági motor

Hibakereső motor

IL → Natív
JIT-fordítók

Kódkezelő

Szemétgyűjtő

Osztálybetöltő

Nyelvek

- A .NET nyelvfüggetlen
 - > A Framework minden szolgáltatása hozzáférhető minden nyelv számára
 - > minden nyelv egyformán fordul, egyik sem interpretált
 - > A különböző nyelveken készült osztályok egymásból örökölhetnek, nincsenek „kiemelt” nyelvek
- Common Language Specification
 - > Közös nyelvi specifikáció
 - > Új, tetszőleges nyelvvel kibővíthető a Framework !

Nyelvek II.

- Compiler platform (Roslyn)
 - > C#, VB, Managed C++, F#
 - > Kiterjeszthető .NET fordító platform
- Egyéb nyelvek:

Common CLI Languages

- **A#**: CLI implementation of [Ada](#).
- **Boo**: A statically typed CLI language, inspired by [Python](#).
- **C#**: Most widely used CLI language, bearing similarities to [Java](#), [Delphi](#) and [C++](#). Implementations provided by [.NET Framework](#), [Portable.NET](#) and [Mono](#).
- **C++/CLI**: A version of [C++](#) including extensions for using [CLR](#) objects. Implementation provided only by [.NET Framework](#). Can produce either [CIL](#)-based managed code or mixed-mode code that mixes both managed code as well as native code. The compiler is provided by Microsoft.
- **Cobra**: A CLI language with both [static](#) as well as [dynamic typing](#), [design-by-contract](#) and built-in [unit testing](#).
- **Component Pascal**: A CLI-compliant [Oberon](#) dialect. It is a strongly typed language in the heritage of Pascal and Modula-2 but with powerful object-oriented extensions.
- **F#**: A multi-paradigm CLI language supporting [functional programming](#) as well as [imperative object-oriented programming disciplines](#). Variant of [ML](#) and is largely compatible with [OCaml](#). The compiler is provided by Microsoft. The implementation provided by Microsoft officially targets both [.NET](#) and [Mono](#).
- **IronPython**: An open-source CLI implementation of [Python](#), built on top of the [DLR](#).
- **IronRuby**: An open-source CLI implementation of [Ruby](#), built on top of the [DLR](#).
- **IronLisp**: A CLI implementation of [Lisp](#). Deprecated in favor of [IronScheme](#).
- **J#**: A [CLS](#)-compliant implementation of [Java](#). The compiler is provided by Microsoft. Microsoft has announced that [J#](#) will be discontinued.
- **JScript.NET**: A CLI implementation of [ECMAScript](#) version 3, compatible with [JScript](#). Contains extensions for [static typing](#). Deprecated in favor of [Managed JScript](#).
- **L#**: A CLI implementation of [Lisp](#).
- **Managed Extensions for C++**: A version of [C](#) targeting the [CLR](#). Deprecated in favor of [C++/CLI](#).
- **Managed JScript**: A CLI implementation of [JScript](#) built on top of the [DLR](#). Conforms to [ECMAScript](#) version 3.
- **Nemerle**: A multi-paradigm language similar to [C#](#), [OCaml](#) and [Lisp](#).
- **Oxygene**: An Object Pascal-based CLI language.
- **P#**: A CLI implementation of [Prolog](#).
- **Phalanger**: An implementation of [PHP](#) with extensions for [ASP.NET](#).
- **Phrogram**: A custom CLI language for beginners and intermediate users produced by [The Phrogram Company](#).
- **PowerBuilder**: Can target CLI since version 11.1.
- **Team Developer**: SQLWindows Application Language (SAL) since Team Developer 6.0.
- **VBX**: A dynamic version of [VB.NET](#) built on top of the [DLR](#). See [VBScript](#) and [VBA](#) as this could be thought of being used like a Managed VBScript (though so far this name has not been applied to this) and could be used to replace VBA as well.
- **VB.NET**: A redesigned, object-oriented dialect of [Visual Basic](#). Implementations provided by [.NET Framework](#) and [Mono](#).
- **Windows PowerShell**: An object-oriented command-line shell. PowerShell can dynamically load .NET assemblies that were written in any CLI language. PowerShell itself uses a unique scripting syntax, uses curly-braces, similar to other C-based languages.

Other CLI languages

- **Active Oberon** - a CLI implementation of [Oberon](#)
- **APLNext** - a CLI implementation of [APL](#)
- **AVR.NET** - a CLI implementation of [RPG](#)
- **clojure-clr** - a CLI implementation of [Clojure](#)
- **Delphi.NET** - a CLI language implementation of the [Delphi](#) language.
- **DotLisp** - a CLI language inspired by [Lisp](#)
- **Delta Forth .NET** - a CLI implementation of [Forth](#) from [Dataman](#)
- **dylan.NET**
- **EiffelEnvision** - a CLI implementation of [Eiffel](#)
- **Fantom** - a language compiling to .NET and to the JVM
- **Fortran.NET** - Fortran compiling to .NET
- **Gardens Point Modula-2/CLR** - an implementation of [Modula-2](#) that can target CIL
- **GrGen.NET** - a CLI language for [graph rewriting](#)
- **IoNET** - a CLI implementation of [Io](#)
- **IronScheme** - a [RGSR](#)-compliant [Scheme](#) implementation built on top of the [DLR](#)
- **IronSmalltalk** - a CLI implementation of [Smalltalk](#) built on top of the [DLR](#)
- **Ja.NET** - an open source implementation of a Java 5 JDK ([Java](#) development tools and runtime) for .NET
- **Common Larceny** - a CLI implementation of [Scheme](#)
- **LOLCode.NET** - a CLI implementation of [LOLCODE](#)
- **Mercury on .NET** - an implementation of [Mercury](#) that can target CIL
- **Net Express** - a CLI implementation of [COBOL](#)
- **NetCOBOL** - a CLI implementation of [COBOL](#)
- **COBOL2002 for .NET Framework** - a CLI implementation of [COBOL](#)
- **COBOL2002 for .NET Framework** - a CLI implementation of [COBOL](#)
- **OxygenScheme** - a CLI implementation of [Scheme](#)
- **PL/I/L** - a CLI implementation of [PL/I](#)
- **#S** - a CLI language that implements [Scheme](#) (a port of [Peter Norvig's Jscheme](#)).
- **#Smalltalk** - a CLI implementation of [Smalltalk](#)
- **sml.net** - a CLI implementation of [Standard ML](#)
- **Synergy.NET** - a CLI implementation of [DIBOL](#)
- **Visual COBOL** - a CLI implementation of [COBOL](#)
- **Vulcan.NET** - a CLI implementation of [xBase Visual Objects](#)
- **X#** - a CLI implementation of [ASM](#) developed for [Cosmos](#). X# was also the codename for the XML-capabilities of [Cw](#).
- **Zonnon**, Yet another CLI-compliant [Oberon](#) dialect.

Felügyelt C++ (managed C++) *

- Létező C++ kód átvitele a .NET Framework környezetébe
- Teljes hozzáférés a .NET Framework-höz
- Még mindig C++
 - > minimális kiterjesztések, az ANSI szellemében
 - a 2005 előtt nagyon csúnya szintaktika
 - > nem vesz el semmit a meglévő lehetőségektől
- Továbbra is „Total Control”
 - > Az egyetlen nyelv, ahol a natív és felügyelt kód, adat keveredhet
 - > Lehetővé teszi a fokozatos átállást C++-ról
 - > Gyors kód integrálása a gyorsan fejlesztett rendszerrel

C#

- A C/C++ család első valódi objektumorientált tagja
 - > nincsenek mutatók, csak referenciák
 - > minden tud, amit egy .NET-es kódban érdemes lehet megvalósítani
 - Tulajdonságok, metódusreferenciák, események, attribútumok, XML dokumentáció
 - > deklaráció és kód egy helyen
 - > minden objektum
 - > a primitív típusokhoz sem kell varázsolni
- A .NET Framework és a Visual Studio nagy része C#-ban készült
- Más nyelvekkel összehasonlítva
 - > Tisztább, mint a C++
 - > Újabb mint a Java, így nagy a hasonlóság a két nyelv között, de „színesebb”, hatékonyabb benne fejleszteni
 - > Egyszerű, mint a VB (vagy Delphi)
 - > Nagyon progresszíven fejlődik: most (2019) a 7.3-as verziótól jár

Felügyelt környezet

Felügyelt adat

+

Felügyelt kód

Ezeket nézzük sorban részletesebben →

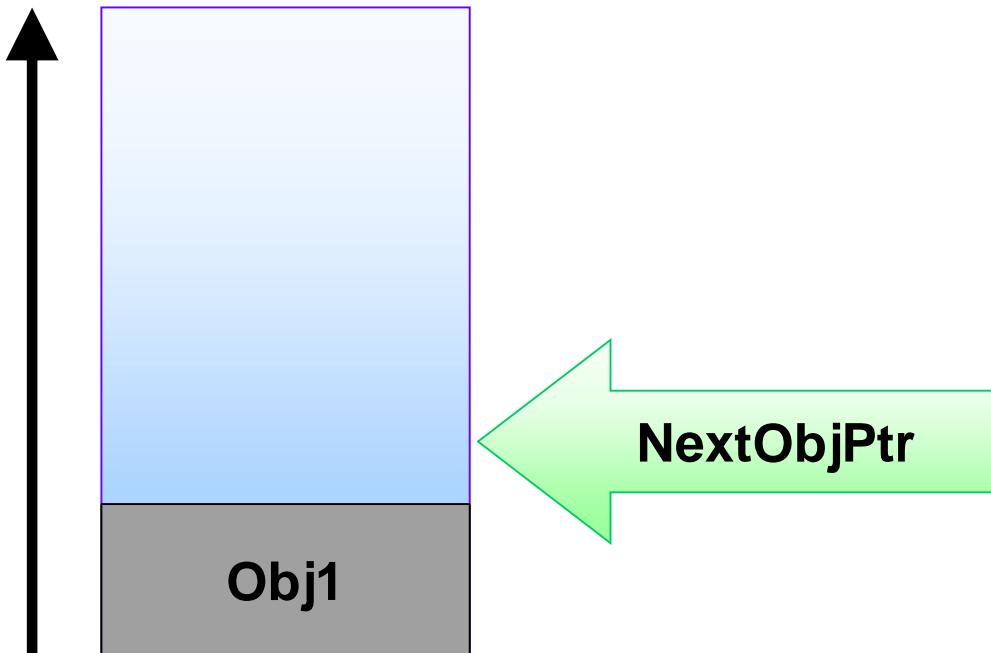
Felügyelt adatok

- Automatikus élettartam-felügyelet
 - > minden .NET objektumot a GC takarít el
 - > Nincs elfelejtett mutató vagy korai *delete* hívás
 - > Hatékony
 - Önhangoló
 - Nem referencia számlált, a hivatkozások bejárásával működik (a körkörös hivatkozásokat is megfejti)
 - > Kétfázisú gyűjtés: *mark & compact*

```
void Process()
{
    MyClass* p = new MyClass;
    p->DoSomething(); // Belül sok, általunk ismeretlen kódot hív
    delete p;
}
```

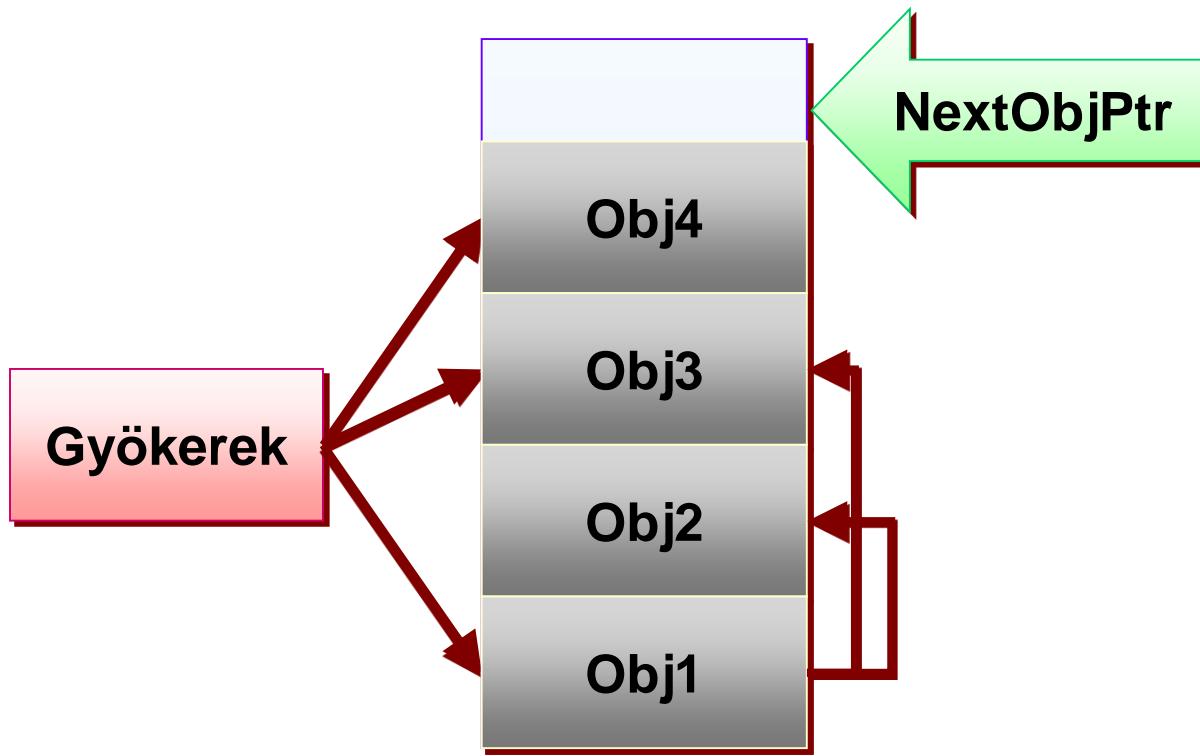
Felügyelt adatok

- Memória foglalása
 - > new: pointer növelése → gyors!



Felügyelt adatok

- Memória felszabadítása
 - > Amikor szükséges (pl. kevés szabad memória): *mark & compact*
 - > Generációs elv:
A fiatal objektumok hamar halnak



1. Gyökerek
2. Elérhetőségi gráf
3. Takarítás
4. Tömörítés
5. Mutatók frissítése

Felügyelt adatok

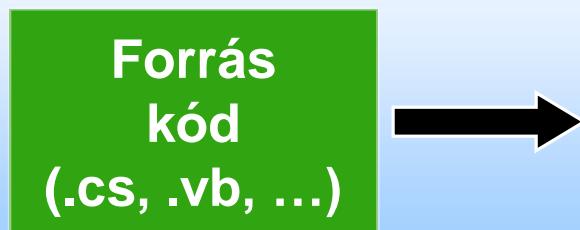
- GC hátrányai
 - > Overhead
 - > Nagy objektumok mozgatása lassú lenne → Large Object Heap (LOH)
 - > Kívülről nem determinisztikus
 - Destruktor mikor fut le?
 - Destruktork milyen sorrendben futnak le?
 - Későbbi előadáson részletesen megnézzük (dispose minta)...
- Ha biztosak vagyunk abban, hogy segít:
`System.GC.Collect();`

Felügyelt kód

- Felügyelt kód: ellenőrizhető
 - > Kötelező metainformációk
 - > Az kód szigorú típusellenőrzésnek vethető alá
 - > **Megszűnnek leggyakoribb hibaforrások**
 - veszélyes típusváltások (type cast, pl. Shape referenciát Rectangle-re castolva, amikor Triangle-re hivatkozik: beszédes kivételt kapunk)
 - inicializálatlan változók
 - tömbből kilógó indexek
 - mutatóbűvész kedések ből eredő hibák
- Egységes kivételkezelés
 - > Nyelvi szinten és a CLR-ben definiált hibakezelés
 - > Integrálódik a Windows strukturált kivételkezelésével is (SEH)
- Biztonsági ellenőrzések

Fordítás és végrehajtás

Fordítás



Nyelvi
fordító



Végrehajtás



JIT fordító



Telepítéskor vagy a
metódus első meghívásakor

Felügyelt kód – nyelvi fordítók

- Közös nyelvi specifikáció (CLS)
- minden nyelvhez más fordító
- Azonos kimenet: IL (Intermediate Language)
- Azonos szolgáltatások:
 - > Nyelvek közti származtatás
 - > Egységes kivételkezelés
 - > Kód ellenőrzés, típus kompatibilitás
 - > Debug szimbólumok

Felügyelt kód – IL

- IL: Intermediate Language
 - > IL alternatív nevek: MSIL, CIL, MIL
- Processzor és architektúra független
- Kiértékelő verem alapú
- Ellenőrizhető
 - > referenciák, típusok, hívási verem, ...
- Továbbfordításra terveztek
 - > Nem interpretált!
- Nyelvfüggetlen
- Objektumorientáltság jellemzi
- Metaadat: típusok leírása, tagváltozók, metódusok leírása
- „Könnyű” visszafejteni, pl. ILSpy - DEMO
 - > Zavarjuk össze, ha ez számít: dotfuscator

.NET Szerelvények

.NET Assembly

Telepítés – szerelvény (assembly)

- Funkciók fizikai egysége
 - > Általában egy .dll vagy .exe fájl (de lehet több is, ritka)
 - > Ez van benne
 - IL kód
 - Metaadatok a szerelvénnyről (manifest)
 - Metaadatok a .NET osztályokról
 - Erőforrások (jpg, .txt, .xml, ...)
 - > Más mint a Java, ahol forrásfájlonként .class fájl
- minden alkalmazás szerelvényekből épül fel
 - > Egy névtér több szerelvényben is lehet
 - > Egy szerelvényben több névtér lehet
 - > Hivatkozhat más szerelvényekre
- Visual Studioban a legtöbb projekt típusnál a projekt kimenete build után egy .exe vagy .dll (kivétel pl. egy webalkalmazás)

Szerelvényszintű metaadat információk (manifest)

- **Név** (általában a fájlnév, kiterjesztés nélkül)
- **Verzió: 4 számjegy - major, minor, build number, revision**
- **Szerelvény referenciaiák** a hivatkozott szerelvényekről:
 - > Név
 - > Verziószám
 - > Nyilvános kulcs részlet, ha azonosított a hivatkozott szerelvény
- Támogatott nyelv és kultúra
- Processzor és OS
- Egyéb attribútumok: kiadó, leírás
- Nyilvános kulcs, ha megosztott
- Hash algoritmus azonosító, ha azonosított
- Szerelvényhez tartozó modulok lista + lenyomat (hash)

A szerelvénny mint egység ...

- Telepítés egysége (verzióval)
- Egymás-melletti végrehajtás egysége
 - > Ugyanabból a szerelvényből több verzió futhat egymás mellett, akár egy folyamaton belül is
- Típus egység
 - > A típusok szerelvényekhez kötődnek és nem névterekhez !
- Biztonsági egység
 - > A kóderedet alapú biztonság esetén (adott helyről letöltött kódnak mire van joga)

Verziókezelés

- Számnégyes: [major].[minor].[buildnumber].[revisionnumber]
- Olyan szemantikát adunk a verzióknak, amit szeretnénk, de próbáljuk értelmesen használni (pl: SemVer szabvány)
 - > Major: általában inkompatibilis verziók
 - teljesen új verzió született
 - > Minor
 - a kompatibilitási szándék megvolt, csak bővült funkciókban
 - > Buildnumber: leggyakrabban kompatibilisek
 - kisebb módosítások – szervíz csomag, hibajavítások
 - biztonságos/normál mód konfigurálható
 - > Revisionnumber: kompatibilis
 - csak hibajavítás
 - hotfix , QFE: Quick Fix Engineering
- AssemblyVersion nevű .NET attribútum határozza meg
 - > Tipikusan AssemblyInfo.cs C# fájlban van megadva
 - > Ne keverjük a csak információ jellegű verzió attribútumokkal

Szerelvény hivatkozási típusok

- **Szerelvények függhetnek egymástól**

- > Pl. egy A.exe használja egy B.dll-ben levő osztályokat
- > Függőség csak egyirányú lehet
 - Ha A.dll használja a B.dll-t, akkor B.dll nem használhatja A.dll-t
 - Így nem alakulhatnak ki függőségi körök (csak függőségi fa van)

- **Szerelvényekre kétféle módon lehet hivatkozni**

- > Privát
- > Azonosított

Most ezt a kettőt nézzük meg alaposabban →

Privát szerelvények - jellemzők

- Egyetlen alkalmazás használja az adott dll-t
- A dll-t
 - > A neve azonosítja
 - > Az alkalmazás mappáiban keresi
 - > Egyszerűen telepíthetőek: xcopy

Azonosított szerelvények

- Szinonimák
 - > Azonosított, strong-named, erős névvel ellátott, megosztott
- Több verzió is egymás mellett (lásd GAC)
- Digitálisan alá kell írni őket a kiadó cég privát kulcsával
- Erős név (strong name) teszi egyedivé, ami ezekből áll:
 - > Név
 - > Fejlesztő cég nyilvános kulcsa
 - > Verziószám
 - > Nyelv és kultúra (opcionális)
- Csak azonosított szerelvényekre hivatkozhat



Szerelvény ütközés
elkerülésére



Szerelvény verziók

Azonosított szerelvények

- Több alkalmazás használhatja az azonosított szerelvényeket
- Szerelvénnytároló, Global Assembly Cache (GAC)
 - > Több alkalmazás által használt szerelvények gyűjtőhelye
 - > Csak azonosított szerelvények lehetnek itt - egyediség, biztonság garantált
 - > %WINDIR%\Assembly mappa alatt
 - .gacutil parancssori eszközzel listázható a tartalma, vagy lehet új szerelvényeket telepíteni
 - Csak a rendszergazda törölhet
 - > Teljesítmény növekedés – ellenőrzés, betöltés

Azonosított szerelvények - verziók

- A klasszikus DLL Hell probléma



- Az „AppB” alkalmazás telepítésével elrontottuk „AppA”-t, mert a „CryptoLib.dll” v2 nem teljesen kompatibilis v1-el.
- DLL Hell problémára megoldást nyújtanak az azonosított szerelvények
 - > Ha telepítünk egy új verziójú DLL-t, akkor a régi alkalmazások a régit használják
 - > A szerelvénnyt kibocsátó (cég) általi verzió átirányítás lehetősége (a javított szerelvénnyt használja az alkalmazást)
 - > A rendszergazdák ezt is felüldefiniálhatják (a kibocsátó cég csak hitte, hogy kijavította a hibát ☺)

Azonosított szerelvények - integritásvédelem

- Integritásvédelem
- Hogyan: a szerelvények digitális aláírása a fejlesztő cég privát kulcsával
 - > A CLR a szerelvény betöltésekor ellenőrzi, nem sérült-e az integritás
 - Nem módosították-e az alkalmazás valamely szerelvényét
 - Nem cserélték-e le az alkalmazás valamely szerelvényét
 - > A szerelvény metaadatoként szerepel
 - A szerelvény aláírása (hash)
 - Az aláíráshoz használt titkos kulcs nyilvános párja (az ellenőrzéshez használt betöltéskor)
 - minden hivatkozott szerelvénnyről részletes információ (ez alapján ellenőrizhető, hogy nem cserélték-e le a hivatkozott szerelvényt)!

NuGet

.NET osztálykönyvtár - System névtér*

- System – alaptípusok, pl. Int32, Array, stb.
- System.Collections
- System.Data
- System.Diagnostics
- System.DirectoryServices
- System.Drawing
- System.IO
- System.Net
- System.Reflection
- System.Security
- System.Text
- System.Threading
- System.Timers
- System.Web
- System.Web.Services
- System.Web.UI, System.Web.UI.HtmlControls, System.Web.UI.WebControls
- System.Windows.Forms
- System.Xml
- és még rengeteg más ...

Irodalom

- .NET, CLR, BCL
 - > www.microsoft.com/net
 - > <https://docs.microsoft.com/en-us/dotnet/>
- C# Programming Guide
 - > Reiter István: C# jegyzet, ingyenes könyv ~C# 5.0-ig
 - > <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/index>

Ellenőrző kérdések

- Ismertesse a felügyelt futtatókörnyezetek fontosabb jellemzőit!
- Ismertesse a Java futtatókörnyezet főbb jellemzőit!
- Ismertesse röviden a .NET verziók (Full, Core, Mono, Standard) jellemzőit!
- Ismertesse a .NET futtatókörnyezet főbb jellemzőit! (CLR, többnyelvűség, stb.)
- Ismertesse a .NET Keretrendszer „Felügyelt adat”-ra vonatkozó szolgáltatásait!
- Ismertesse a .NET Keretrendszer „Felügyelt kód”-ra vonatkozó szolgáltatásait!
- Ismertesse a fordítás és végrehajtás lépéseit!
- Ismertesse a .NET szerelvény fogalmát!
- Jellemezze a .NET IL kódot!
- Ismertesse a .NET szerelvények két fő csoportját és adja meg ezek jellemzőit!
- Ismertesse a DLL Hell problémát, és hogy milyen megoldást nyújt erre a .NET!
- Mit jelent a .NET az alkalmazások integritásvédelme? Hogyan támogatja ez a .NET keretrendszer?

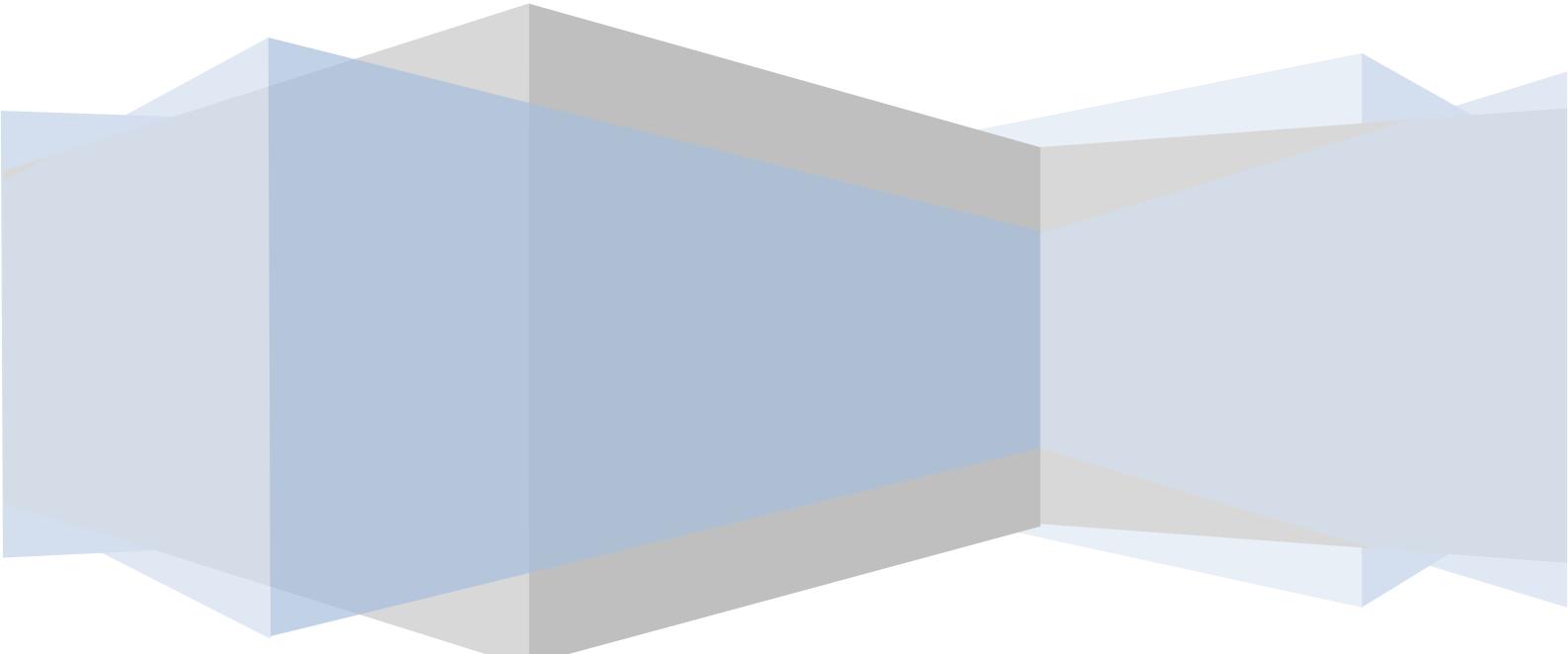
BME AAIT

Modern programozási eszközök

Szoftvertechnikák

Benedek Zoltán

2019.02.13



Tartalom

Contents

Tartalom.....	2
Property (tulajdonság)	3
Delegate (delegát, metódusreferencia).....	5
Event (esemény)	7
Attribute.....	9
Érték és referencia típusok	10

Property (tulajdonság)

Az osztályaink tagváltozót általában nem célszerű publikussá tenni, mert akkor nem garantálható az objektumok konzisztens állapota (pl. a születési év tagváltozónak negatív érték is adható). Erre klasszikus megoldás az, ha a tagváltozókat védetté tesszük, majd lekérdezésükhez egy getXXX, szabályozott beállításukhoz egy setXXX tagfüggvényt vezetünk be. A C# nyelvben erre egy szintaktikailag egyszerűbb megoldás is van, ennek neve property (tulajdonság). A főbb jellemzők a következők (a következő példa alapján):

- A definíciója a tagváltozókhoz hasonló, de megadunk egy get {} és egy set {} ágat is.
- A get {} ág a tulajdonság *lekérdezésekor* fut le. Ha nem írunk get{} ágat, akkor a tulajdonság nem kérdezhető le.
- A set {} ág a tulajdonság állításakor fut le. Ha nem írunk set{} ágat, akkor a tulajdonság nem állítható be.
- A tulajdonság egy közönséges nyelvi elem lett, osztályoknak most már nem csak tagváltozói és tagfüggvényei lehetnek, hanem tulajdonságai is.
- Egy objektum adott tulajdonságának elérése olyan szintaktikával történik, mintha az objektum tagváltozója lenne.

```
class Person
{
    private string name;
    private int yearOfBirth;

    // Name nevű property bevezetése
    public string Name
    {
        get { return name; }
        set
        {
            if (value == null)
                throw new ArgumentNullException("The Name property can not be null.");
            name = value;
        }
    }

    // YearOfBirth nevű property bevezetése
    public int YearOfBirth
    {
        get { return yearOfBirth; }
        set
        {
            if (value < 1800 || value > 5000)
                throw new ArgumentException("Invalid yearOfBirth value.");
            yearOfBirth = value;
        }
    }

    // Számított érték, csak olvasható property.
    public int Age
    {
        get { return DateTime.Now.Year - yearOfBirth; }
    }

    // Ez kell ahhoz, hogy konzisztens legyen!
    // A property-n keresztül állítjuk, hogy meglegyen a validáció.
    public Person(string myName, int yearOfBirth)
    {
        Name = name;
```

```

        YearOfBirth = yearOfBirth;
    }

}

class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person("Béla", 1980);
        p1.YearOfBirth = 1995; // set-et hív
        Console.WriteLine("Név: {0}, kor: Age:{1}", p1.Name, p1.Age); // get-et hív

        p1.Age = 20; // Fordítási hiba, nincs set
        p1.YearOfBirth = 1000; // Futás közbeni hiba
        ...
    }
}

```

A property minden .NET nyelvben elérhető, nem csak C#-ból (csak más a szintaktika).

Auto-implemented property

.NET-ben gyakran fordul elő, hogy olyan property-t készítünk, mely lekérdezéskor pusztán visszaadja egy tagváltozó értékét, beállításkor egyszerűen minden validáció nélkül beállítja azt. Pl.:

```

class Person
{
    public string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}

```

Jelen tudásunk alapján még nem érhető, mi értelme lehet ennek, hiszen az egészet kiválthatnánk egy normál publikus tagváltozó használatával. Bizonyos .NET „modulok” viszont kifejezetten építenek a property-k meglétére (pl. Form-ok estében az adatkötés), így mégiscsak nem is ritkán találkozunk hasonlóval. C# 3.0-tól kezdve lehetőség van auto-implementált property írására, amivel a fenti példa tömörebb formába hozható:

```

class Person
{
    public string Name
    {
        get; set;
    }
}

```

Ha a get és a set kulcsszavak után nem adunk meg implementációt, akkor auto-implementált property keletkezik. Az osztály a property-ez generál egy láthatatlan (kódjából nem is elérhető) tagváltozót, és a property lekérdezésekor ennek értékét adja vissza, illetve ezt állítja. További lehetőségek:

- Lehetőség van csak get vagy csak set megadására.
- Arra is lehetőségünk van, hogy a get vagy a set ágra vonatkozóan szigorítsuk a láthatóságot. Pl. az alábbi osztályban a property-t a külvilág csak olvasni tudja, míg a saját metódusai írni is:

```
class Person
{
    public string Name
    {
        get;
        private set;
    }
}
```

Delegate (delegát, metódusreferencia)

Olyan, mint a C-ben a függvénypointer, csak objektumorientált, illetve a C-vel szemben nemcsak egy, hanem több függvényre (metódusra) is lehet vele mutatni (hivatkozni). A delegate-ek használatának egyik előnye az, hogy futási időben dönthetjük el, hogy több metódus közül éppen melyiket szeretnénk meghívni.

A delegate-ek (hasonlóan a C függvénypointerekhez) **típusosak**, egy delegate objektummal a típusának megfelelő szignatúrájú és visszatérési értékű metódusra lehet hivatkozni. Amikor delegate-ekkel dolgozunk, első lépésben egy delegate típust kell definiálni. Ez annak felel meg, mint amikor C-ben a typedef kulcsszóval egy függvénypointer típust definiáltunk. Ennek megfelelően egy **delegát típus** definiálásával egy olyan típust definiálunk, amelynek változóival rámutathatunk egy vagy több olyan metódusra, amely kompatibilis (paraméterlistája és visszatérési típusa) a delegát típusával. Pl.:

```
delegate bool FirstIsSmallerDelegate(object a, object b);
```

Itt a FirstIsSmallerDelegate egy delegate típus. Ebből pont úgy hozhatunk létre változókat (lokális, tagváltozók), vagy szerepeltethetjük függvényparaméterben, mintha egy közönséges osztály lenne, pl.:

```
FirstIsSmallerDelegate fis1;
```

Itt a fis1 delegate változó (objektum) értéke null. Értéket így adhatunk neki:

```
fis1 = new FirstIsSmallerDelegate(FirstIsSmaller_Complex);
```

A delegate típus „konstruktornak” a visszahívandó függvény nevét kell megadni. Itt feltettük, hogy a hívó kód osztályában létezik egy FirstIsSmaller_Complex nevű olyan függvény, amely kompatibilis a FirstIsSmallerDelegate delegate típussal (van két object paramétere és bool-lal tér vissza.)

Amikor értéket adunk egy delegate objektumnak, lehet az egyszerűsített szintaktikát is használni, amikor csak a függvény nevét adjuk meg:

```
fis1 = FirstIsSmaller_Complex;
```

A delegát meghívásával a delegate objektum által hivatkozott metódus automatikusan meghívódik:

```
bool res = fis1(obj1, obj2); // meghívódik a FirstIsSmaller_Complex
```

A delegátok használatának egyik előnye az, hogy futási időben dönthetjük el, hogy több metódus közül éppen melyiket szeretnénk meghívni.

Példa

Az alábbi példában a **Sorter** osztály **HyperSort** függvénye egy általános sorrendező művelet. Tetszőleges típusú elemeket tud sorrendezni. Ehhez paraméterként megkapja az elemlistát, valamint az elemeket összehasonlítható képes metódusreferenciát.

```
class Complex
{
    public double Re, Im;
```

```

// Konstruktor
public Complex(double re, double im)
{
    this.Re = re;
    this.Im = im;
}

delegate bool FirstIsSmallerDelegate(object a, object b);

class Sorter
{
    // A lista rendezése egy paramétkent kapott delegate segítségével.
    // tetszőleges típusra használni szeretnénk. A Complex forráskója
    // nincs meg, nem tudjuk megoldani, hogy implementálja az
    // IComparable-t. Megoldás: átadjuk az összehasonlító "függvényt" is.
    public static void HyperSort(ArrayList list,
        FirstIsSmallerDelegate firstIsSmaller)
    {
        for (...)
        {
            ...
            if ( firstIsSmaller(list[j], list[j - 1]) )
                ...
        }
    }
}

class Program
{
    static void Main(string[] args)
    {
        ArrayList list = new ArrayList();
        list.Add(new Complex(1, 2)); // Egy elem, biztosra megyünk :).
        // ...

        // Metódusreferencia statikus tagfüggvényre
        Sorter.HyperSort(list, new FirstIsSmallerDelegate(FirstIsSmaller_Complex));
        // Egyszerűsített szintaktikával is lehet, ekkor csak a függvény nevét írjuk ki
        Sorter.HyperSort(list, FirstIsSmaller_Complex);

        // Metódusreferencia objektum tagfüggvényre (kicsit erőltetett,
        // itt ez is lehetne statikus
        Comparers comps = new Comparers();
        Sorter.HyperSort(list, new FirstIsSmallerDelegate(comps.FirstIsSmaller_Complex));

        // A delegate egy típus, lehet lokális változó, tagváltozó is.
        FirstIsSmallerDelegate fis1 = new FirstIsSmallerDelegate(FirstIsSmaller_Complex);
        bool isFIS = fis1(new Complex(1, 1), new Complex(2, 2));

        // minden delegate egy MultiCastDelegate leszármazott. Több metódusreferenciát
        // is tud tárolni. A += operátorral vehetők fel újak. Az alábbi példában kétszer
        // is meghívódik a FirstIsSmaller_Complex, nincs sok értelme. Majd az event-eknél
        // látjuk, miért jó ez.
        fis1 += FirstIsSmaller_Complex;
        fis1(new Complex(1, 1), new Complex(2, 2));
    }

    public static bool FirstIsSmaller_Complex(object a, object b)
    {
        Complex ca = (Complex)a;

```

```

        Complex cb = (Complex)b;
        return Math.Sqrt(...);
    }

class Comparers
{
    public bool FirstIsSmaller_Complex(object a, object b)
    {
        // mint a Program osztályban
        --| |--
    }

    public bool FirstIsSmaller_Person(object a, object b)
    {
        ...
    }
}

```

Egy másik megközelítés az lehet, ha nem metódusreferenciát használunk, hanem az sorrendezendő objektumok típusának kell egy interfész implementálniuk (pl. `IComparable`), ami két elem összehasonlítását elvégzi. Az olyan nyelvek esetében, melyek nem támogatják a függvénypointer/metódusreferencia koncepcióját, ezt szokás használni. .NET környezetben mindenki megoldás használható, az esettől függ, melyiket célszerűbb választani. Az interfész alapú megközelítés kicsit „egységezűbb” megközelítést jelent, a delegate alapú kicsit rugalmasabb:

- Lehet statikus függvényre is metódusreferencia, nem kell hozzá objektum.
- Akkor is használható, ha nem tudjuk megoldani, hogy implementálja az adott interfész, pl. „`IComparable`”-t.

Event (esemény)

Az alkalmazások többsége manapság már eseményvezérelt. Ez azt jelenti, hogy az alkalmazás bizonyos elemei eseményeket váltanak ki bizonyos sorrendben (például egy gomb megnyomása a felhasználói felületen), amelyekre az alkalmazás más részei reagálnak.

Az eseménykezelés .NET-ben, mint ahogy általában minden más programozói nyelvben (feltéve, hogy támogatja azt), a **Publisher/Subscriber tervezési mintára épül**. Ennek lényege, hogy **tetszőleges osztály publikálhatja eseményeknek egy csoportját, amelyekre tetszőleges más osztályok objektumai előfizethetnek**. Amikor a publikáló osztály objektuma elsüti az eseményt, az összes feliratkozott objektum értesítést kap erről. Megjegyzés: bár sokkal ritkább, lehetőség van osztály szinten is eseményeket definiálni (statikus esemény), illetve eseményekre osztály szinten feliratkozni.

.NET-ben az eseménykezelés hátterében a delegátorok állnak. Az eseményt publikáló osztály tulajdonképpen egy multicast delegátból egy speciális tagváltozót definiál az **event** kulcsszóval, amely a feliratkozott osztályok egy-egy metódusára mutat. Az event egy közönséges nyelvi elem lett, osztályoknak most már nem csak tagváltozói, tagfüggvényei és tulajdonságai lehetnek, hanem eseményei is.

Amikor az esemény elsül, a feliratkozott osztályok megfelelő metódusai a delegáton keresztül meghívódnak. A feliratkozott osztályok azon metódusait, amelyekkel az egyes eseményekre reagálnak, eseménykezelő metódusoknak nevezzük.

Minden eseménynek van egy típusa és van egy neve. A típusa annak a delegátnak a típusa, amelyik tulajdonképpen megfelel magának az eseménynek.

Példa

Az alábbi példában a **Logger** egy általános célú naplózó osztály. Ez egy event-tet definiál **Log** néven. A Logger osztály esetében naplózni a WriteLine függvényel lehet, ami nem csinál mást, mint elsüti a Log eseményt (a null vizsgálattal előbb megvizsgálja, van-e legalább egy előfizető). Az **App** osztály az alkalmazást reprezentálja. A konstruktőrben két előfizető műveletet is regisztrál a **+= operátorral**: a saját **writeConsole** tagfüggvényét (ami a konzolra naplóz), valamint egy **FileLogListener** objektum **WriteToFile** tagfüggvényét (fájlba naplóz). Eseményről leíratkozni a **-= operátorral** lehet (lásd **App.Cleanup** művelet).

```
public delegate void LogHandler(string msg);

class Logger
{
    // Osztálynak .NET-nem nem csak tagváltozója és tagfüggvénye
    // lehet, hanem event-je is!
    public event LogHandler Log;
    // Ide jöhet a többi, de most nincs több

    public void WriteLine(string msg)
    {
        // Esemény elsütése (null vizsgálat: meg kell nézni, van-e előfizető)
        if (Log != null)
            Log(msg);
    }
}

class FileLogListener
{
    // FileStream tag.
    public void WriteToFile(string msg)
    {
        // ...
    }
}

class App
{
    Logger log = new Logger();
    FileLogListener fileLogListener = new FileLogListener();

    public App()
    {
        // Feliratkozás a Log eseményre (a writeConsole és fileLogListener.WriteToFile
        // műveletek regisztráljuk be)
        log.Log += new LogHandler(writeConsole);
        log.Log += new LogHandler(fileLogListener.WriteToFile);

        // Egyszerűsített formával is feliratkozhatunk, csak a kezelőfüggvény nevét
        // adjuk meg. A fenti két sorral ekvivalens:
        log.Log += writeConsole;
        log.Log += fileLogListener.WriteToFile;
    }

    public void Process()
    {
```

```

        log.WriteLine("Process begin...");
        //...
        log.WriteLine("Process end...");
    }

    public void Cleanup()
    {
        // Leíratkozás eseményről
        log.Log -= new LogHandler(writeConsole);
        log.Log -= new LogHandler(fileLogListener.WriteToFile);
    }

    void writeConsole(string msg)
    {
        Console.WriteLine(msg);
    }
}

class Program
{
    static void Main(string[] args)
    {
        App app = new App();
        app.Process();
        app.Cleanup();
    }
}

```

Miben más az event, mint a delegate?

- Egy delegate objektumból akkor lesz event, ha elő írjuk az event kulcsszót.
- Az event osztályok tagváltozója lehet csak (így pl. lokális event objektum nincs, lokális delegate objektum létezik viszont.)
- Nem lehet az = operátort használni, csak a += és -= (így egy külső objektum nem tudja kitörölni a feliratkozottakat a listáról)
- Csak a tartalmazó osztály sútheti el.

Az események minden .NET nyelvben elérhetők, csak más szintaktikával.

Attribute

Az attribútumok segítségével deklaratív jelleggel metaadatokat közölhetünk a kód bizonyos részeire vonatkozóan. Az attribútum is tulajdonképpen egy osztály, amit hozzákötnünk a program egy megadott eleméhez (típushoz, osztályhoz, interfészhez, metódushoz, ...). Ezeket a metainformációkat a program futása közben mi magunk is kiolvashatjuk az úgynévezett reflection mechanizmus segítségével (ezzel részletesen egy későbbi gyakorlat foglalkozik), de általában az attribútumokkal a .NET „beépített” osztályai számára szeretnénk információkat közölni. A .NET attribútumoknak a Java nyelvben az annotációk felelnek meg.

Az attribútumok funkciója a legkülönbözőbb féle lehet. A Serializable attribútum segítségével például egy osztályról jelezhetjük, hogy az binárisan sorosítható, azaz tetszőleges adatfolyamba (akár egy file-ba, akár egy hálózati adatfolyamba) az állapota bináris formában elmenthető:

```

[Serializable] // Jellezzük, hogy az osztály sorosítható
class User
{

```

```

    string name;

    [NonSerialized] // Jelezzük, hogy ezt a mezőt nem kell sorosítani
    string password;

    // Jelezzük a jogosultsági feltételeket
    [PrincipalPermission(SecurityAction.Demand, Role = "Admin")]
    public static void DeleteUser(int userId)
    {
    }

    // ...
}

class Program
{
    static void Main(string[] args)
    {
        User user = new User();
        // felparaméterezzük ...

        // Szerializálás egy file stream-be
        BinaryFormatter formatter = new BinaryFormatter();
        FileStream stream1 = new FileStream("Dump.dat", FileMode.Create);
        formatter.Serialize(stream1, user);
        stream1.Close();

        // Deszerializálás a file stream-ból
        FileStream stream2 = new FileStream("Dump.dat", FileMode.Open);
        User u = (User)formatter.Deserialize(stream2);
        stream2.Close();
    }
}

```

Tudunk olyan kódot írni, amivel lekérdezhetjük, hogy :

- a User osztálynak milyen attribútumai vannak
- a User osztálynak milyen tagváltozói/tagfüggvényei vannak, ezeknek milyen attribútumai (a BinaryFormatter is ezt csinálja, valamint a jogosultságellenőrző is).

Érték és referencia típusok

.NET környezetben minden típus vagy az érték vagy a referencia típusok csoportjába tartozik, és ennek megfelelően viselkedik.

A lényegi jellemzők a következők:

- **Érték típus (value type):** A változó magát az értéket tartalmazza (nem egy mutató rá). int, char, decimal, double, float, bool, enum, stb. egyszerű típusok tartoznak ide, meg amit mit definiálunk a **struct** kulcsszóval. Inline módon foglalnak helyet összetett típusokban. Lokális változónál a vermen foglalódik nekik hely. Függvényparaméter átadáskor pont úgy kezelődnek, mint C++-ban az int, vagy minden más: másolat készül az eredeti adatról. Gyors az allokációjuk. Korlátozások: nem örökölhetnek, nem lehet belőlük származni. Interfészt viszont implementálhatnak (lásd később).
- **A referencia típus (reference type):** két részből áll: egy mutató/hivatkozás, ami alapértelmezésben null, és maga a mutatott/hivatkozott objektum, ami a felügyelt heapen foglal helyet, és a garbage collector gyűjti be, ha már nincs rá hivatkozás. Ez utóbbinak nekünk kell a

new-val helyet foglalni. A .NET beépített osztályai (pl. string, File, stb.) ilyenek, a tömbök, meg amit mi hozunk létre a **class** kulcsszóval. Az interfészek is ide tartoznak, később lesz róluk szó.

Vagyis ha egy saját típust hozunk létre a class kulcsszóval, akkor az alapvetően úgy viselkedik, mint Java-ban az osztályok. A beépített egyszerű típusok (pl. int, char, stb.), valamint az általunk létrehozott struct típusok viszont úgy viselkednek, mint a C/C++ egyszerű típusai (illetve az összetett típusok is, amikor nem használunk referenciát/pointert).

Modern nyelvi eszközök

.NET - C# nyelv elemei

Szoftvertechnikák 2. előadáshoz kiegészítő anyag

Automatizálási és Alkalmazott Informatikai Tanszék, BME

Áttekintés

- **Jelen diasorozat egy tömör áttekintést ad a fontosabb újszerű C# nyelvi elemekről. Számos témakört érint, amely nem tananyag, hanem érdekességként szerepel. A számon kért témakörök a következők:**
 - ◆ Property (tuljadonság)
 - ◆ Delegate (metódusreferencia)
 - ◆ Event (esemény)
 - ◆ .NET attribute (.NET attribútum)
 - ◆ Érték és referencia típusok
 - ◆ Be- és kidobozolás
- **Ezek jó részét sokkal részletesebben, példákon keresztül ismerteti az „Előadás 02 - Modern nyelvi eszközök.pdf” dokumentum.**

Tartalom

- Bevezető
- **C# elemei és eszközei**
 - ◆ a fájlok struktúrája
 - ◆ a C# típusrendszere
 - ◆ a C# nyelv elemei

Bevezető (1)

- **Nagyon szép**
 - ◆ minimális tanulás (a C-hez, Java-hoz képest)
 - ◆ olvasható
- **Első komponens orientált nyelv a C/C++ nyelvcsaládban (a Java után)**
 - ◆ namespace
 - ◆ verziók követése
 - ◆ attribútumok támogatása
- **Erőteljesebb, mint a Java**
- **Sokkal tisztább, mint a C++**
- **A C++ ereje, a VB (Delphi) könnyedségével ötvözve**

Bevezető (2)

- Illeszkedjen a .NET framework-höz
- Minimalizáljuk a fejlesztést és a hibalehetőségeket
- A gyakori és hasznos funkciókat emeljük be a nyelvbe
- Hasznosítsuk az elméleti eredményeket
- Biztonságos kód

C# program struktúra

```
// A skeleton of a C# program
using System;

namespace MyNamespace1 {

    class MyClass1 { }

    struct MyStruct { }

    interface IMyInterface { }

    delegate int MyDelegate();

    enum MyEnum { }

    namespace MyNamespace2 { }

    class MyClass2 {

        public static void Main(string[] args) { }
    }
}
```

Szia világ !

```
namespace Sample
{
    using System;

    public class HelloWorld
    {
        // nem feltétlenül szükséges
        public HelloWorld()
        {
        }

        public static int Main(string[] args)
        {
            Console.WriteLine( „Szia világ!“ );
            return 0;
        }
    }
}
```

Hello World! - anatómia

- Saját névterében van a példaprogram
- Más névtereket használ a „using” kulcsszóval
- Publikus osztály,
- Publikus belépési pont
 - "static int Main(...)"
- Kiírja a „Szia világ!” szöveget a konzolra
 - ◆ a System.Console osztály statikus metódusát használja
- A konstruktur nem feltétlenül szükséges

A fájlok struktúrája

- **Nincs header fájl**
- **C# támogatja a „definíció a deklarálásnál” modellt**
 - ◆ mint a Visual Basic, Pascal, Modula, Java
- **Minden kód és deklaráció egy helyen**
 - ◆ a kód konzisztens és könnyen kezelhető
 - ◆ csapatmunka esetén sokkal közérthetőbb
 - ◆ deklaráció metaadatokon át jobban elérhető
- **Feltételes fordítás (preprocessor) van, de makró nincs (#define, #if)**
 - ◆ #if-nél tisztább

```
[Conditional ("CONDITION1") ]  
public static void Method1() {...}
```

A típusrendszer áttekintése

- A C# típusrendszere .NET CLS típusrendszerére (CTS-re) épül
- C#-ban natív elérés a .NET típus rendszerhez
 - ◆ C# a .NET bölcsőjéből született
 - ◆ Pl. CTS: System.Int32 →C#: int
- minden Objektum
 - ◆ minden objektum *implicit* a System.Object-ből származik

```
Console.WriteLine( 256.ToString() );
```
- Megkülönböztetés érték és a referencia típusok között
 - ◆ Érték: egyszerű típusok, enum, struct
 - ◆ Referencia: interface, class, array, string, ...

Beépített típusok

■ Egész típusok

- ◆ byte, sbyte (8 bit)
- ◆ short, ushort (16 bit)
- ◆ int, uint (32 bit)
- ◆ long, ulong (64 bit)

■ Boolean típus

- ◆ bool (külön típus, nem cserélhető fel az int-tel)

■ IEEE lebegőpontos típusok

- ◆ double (32 bit, pontosság 15-16 számjegy)
- ◆ float (16 bit, pontosság: 7 számjegy)

■ Pontos szám típus

- ◆ decimal (128 bit, 28 szignifikáns számjegy)

■ Karakter típus

- ◆ char (1 db unicode16 karakter, 16 bit)

■ String

- ◆ string (megváltoztathatatlan unicode16 szöveg), referencia típus

System.Object

- **Minden típus implicit vagy explicit a System.Object-ból származik**
- **C#-ban: object**
- **Metódusai**
 - ◆ virtual string `ToString()`: szöveges reprezentáció
 - ◆ virtual int `GetHashCode()`: hash érték generálása
 - ◆ virtual bool `Equals(object o)`: az objektum azonos-e egy másikkal
 - ◆ Type `GetType()`: futás idejű típusreprezentáció
 - ◆ Néhány egyéb:
 - static bool `Equals(object o1, object o2)`
 - static bool `ReferenceEquals(object o1, object o2)`
 - protected object `MemberwiseClone()`
 - protected `Finalize()` – C#-ban destruktur

Érték és referencia típusok

- **Érték típusok (value types)**
 - ◆ A vermen jönnek létre
 - Automatikusan törlődnek a veremről
 - ◆ Az összetett típusokban „in-line” foglalnak helyet
 - ◆ Mindig másolódnak
 - A változók az értéket tartalmazzák
 - ◆ int, char, decimal, float, bool, enum, struct
- **Referencia típusok (reference types)**
 - ◆ A heap-en jönnek létre
 - A GC takarítja el őket a memóriából
 - ◆ Referencia szerint adódnak át
 - A változó csak egy referenciát (mutatót) tartalmaz
 - ◆ string, tömbök, class, interface, ...

Referencia típusok

- **A GC takarítja el őket**
 - ◆ Nagy teljesítmény csökkenés lehet
 - ◆ Ne hívunk GC.Collect-et (önhangoló)
- **SOHA ne használjunk destruktort (finalizert)**
 - ◆ Csak akkor, ha nem-felügyelt erőforrást használnak
- **A nagy objektumokat mielőbb engedjük el**
 - ◆ A referenciát null-ozzuk ki
- **Nehogy megfogjunk valamit, ami nem kell**

Érték típusok

System.Object

System.ValueType

System.Int32

- **Másolódnak**
 - ◆ Nem „egy példány” van belőlük
 - ◆ A méret számít, kis objektumok előnyben
- **Nem kell GC**
 - ◆ Nagyon jelentős teljesítménynövekedés lehet
 - MÉRNI, MÉRNI, MÉRNI !!!
- **Korlátozások**
 - ◆ Nem örökölhetnek
 - ◆ Nem lehet belőlük örökölni
 - ◆ Interfész implementálhatnak
- **Például**
 - ◆ **DateTime**, **Decimal**, **enum**
 - ◆ **Complex** számok, 2 és 3 dimenziós koordináták
 - ◆ deviza típus

System.String

- C#-ban: string
- Egy string típusú változó tartalma (a szöveg) nem változtatható meg
 - ◆ minden string művelet egy új példányt ad vissza. Pl.:
„alma”.Replace(„m”, „b”)
s += “hello”;
s[1] = “a”; // hibás, csak olvasni lehet így
 - ◆ A régit a GC-nek kell eltakarítania !
- Használunk System.Text.StringBuilder-t
 - ◆ vagy System.IO.StringWriter-t, stb.
- Minimalizáljuk a string műveleteket !

Statementek

■ C-szerű: Flow Control és Loop

- ◆ `if (<bool expr>) { ... } else { ... }`
- ◆ `switch(<var>) { case <const>: ...; }`
- ◆ `while (<bool expr>) { ... }`
- ◆ `for (<init>;<bool test>;<modify>) { ... }`
- ◆ `foreach(típus <var> in <var>) { ... }`
- ◆ `do { ... } while (<bool expr>);`

■ Nem C szerű:

- ◆ `lock(<object>){ ... };`
 - nyelvvel járó kritikus szekció szinkronizáció lehetőség
- ◆ `checked { ... }; unchecked { ... };`
- ◆ `checked (...); unchecked (...);`
 - uncheck: aritmetikai műveletek hibáinak kezelése, nullával osztás NEM

```
switch( n )
{
    case 0:
    case 3:
        Console.WriteLine( "3 vagy 0 vagy 9" );
        break;

    case 4:
        Console.WriteLine( "4" );
        // hiba! nincs break vagy goto
    case 5:
        Console.WriteLine( "4 vagy 5" );
        break;

    case 9:
        goto case 3;

    default:
        Console.WriteLine( "más" );
        break;
}
```

A felsorolt típus (enum)

- Nevesített elemek használata a számok helyett
- Erősen típusos
- Jobban használható a "Color.Blue" mint a "3"
 - ◆ Jobban olvasható, könnyebben kezelhető
 - ◆ Még mindig olyan „könnyű”, mint az int
- Tetszőleges integer típus használható
- Bitmezők (Flags)

Enum példa

```
enum Color : byte
{
    Red,          // = 0
    Green,        // = 1
    Blue          // = 2
}
...
Color c = Color.Blue;
Console.WriteLine( c.ToString() ); // Blue
Console.WriteLine( c.ToString("D") ); // 2
Color[] colors =
    (Color[])Enum.GetValues(typeof(Color));
foreach( Color c in colors )
    Console.WriteLine( "{0:G} = {0:D}", c );
    // Red = 0
    // ...
```

Enum példa

[Flags]

```
enum Color : byte
{
    Red = 0x0001,
    Green = 0x0002,
    Blue = 0x0004
}
...
Color white = Color.Red | Color.Green |
    Color.Blue;
Console.WriteLine( white.ToString() );
    // Red, Green, Blue
```

Osztályok

- **Kód és adat implementációja**
- **Több interface-t implementálhat**
- **Egy osztálytól örökölhet**
- **Osztályok tartalmazhatnak:**
 - ◆ mezők: tagváltozó
 - ◆ tulajdonságok: érték elérése get/set metódus párokon át, lehet csak olvasható is !
 - ◆ Metódusok (statikus is): funkcionálitás
 - ◆ speciális: event, indexer, delegate, ...
- **Fizikai struktúra (assembly)**
- **Logikai struktúra (namespace)**

Tulajdonságok (property-k)

- Mezőeléréshez funkció rendelése
- Felhasználása
 - ◆ csak olvasható tagok implementációja
 - ◆ hozzárendelés validációja
 - ◆ számolt vagy képzett értékek
 - ◆ felfedni interfészen keresztül örökölt értékeket
- Ha komponensnek van: megjelennek a designerben

```
string Name
{
    get { return name; }
    set { name = value; }
}
```

Interface-ek

- Metódusok (és tulajdonságok, eventek, indexerek) absztrakt definíciója
 - ◆ Komponens alapú gondolkodás
- Struktúra és szemantika definiálása speciális célra
- Támogatja a többszörös interfész implementációt

```
interface IPersonAge
{
    int YearOfBirth {get; set;}
    int GetAge();
}
```

Class, Interface, Property példa

```
public class Person : IPersonAge
{
    private int YOB = 0;
    public int YearOfBirth
    {
        get { return YOB; }
        set { YOB = value; }
    }
    public int GetAge()
    {
        return Now.Year - YearOfBirth;
    }
}
```

Class, Interface, Property példa

```
public class Person  
{  
    public int Age;  
}  
...
```

```
Person p = new Person();  
p.Age = 33;  
...
```

```
Console.WriteLine( "személy kora: {0}",  
    p.Age );
```

Class, Interface, Property példa

```
public class Person  
{  
    public int Age;  
    public int Birth;  
}
```

?

Class, Interface, Property példa

```
public class Person
{
    private int age;
    public int Age
    {
        get { return age; }
        set { age = value; birth = DateTime.Now.Year - age; }
    }
    private int birth;
    public int Birth { get { return birth; } }
}
...
Person p = new Person();
p.Age = 33;
...
Console.WriteLine( "személy kora: {0}", p.Age );
```

Indexerek

- Konzisztens mód a containerek építésére
- A „tulajdonságok” ötleten alapszik
- Indexelt elérést biztosít a tartalmazott objektumokhoz
- Az index minősítő bármilyen típusú lehet

```
object this[string index]
{
    get { return Dict.Item( index ); }
    set { Dict.Add( index, value ); }
}
```

Operátorok

■ C-szerű:

- ◆ Logikai: && || ^ !
 - ◆ Aritmetikus: * / + - % << >>
 - ◆ Relácionális: == != < > >= <=

■ C-hez hasonló:

- ◆ Integer: & és | bináris AND/OR
 - ◆ Bool: & és | logikai operátor teljes kiértékeléssel
 $1 == 0 \&\& isValid(a)$
 $1 == 0 \& isValid(a) \Rightarrow isValid$ hívás

■ Nem C-szerű:

- ◆ **is:** Runtime-Type tesztelése
 - ◆ **as:** Type-Cast, nincs exception, null lesz
 - ◆ **typeof:** Runtime-Type lekérése, System.Type-al tér vissza

Operátor felüldefiniálás

- **Legtöbb operátor felüldefiniálható**
 - ◆ aritmetikai, relációs és logikai operátorok
- **Nem felüldefiniálható:**
 - ◆ Hozzárendelés operátor (=)
 - ◆ Speciális operátorok (`sizeof`, `new`, `is`, `typeof`)

```
static Total operator + ( int Amount, Total t )
{
    t.total += a;
    return t;
}
```

Operátor felüldefiniálás a frameworkben

- **Decimal, DateTime, TimeSpan**
- **Point, Rectangle**
- **SQLString, SQLInt, ...**

Delegate

- **Függvény pointerek**
 - ◆ statikus metódusokra
 - ◆ tagfüggvényekre
- **Alapelv:**
 - ◆ Egy statikus műveletet vagy egy adott objektum adott műveletét egy delegate objektumba csomagoljuk.
 - ◆ Ezután a delegate objektum továbbadható, eltárolható mezőbe stb.
 - ◆ A delegate objektumon keresztül bármikor meg lehet hívni a becsomagolt függvényt.
- **Erősen típusos, nincs type-case zavar**
 - ◆ nem kell típuskonvertálni
 - ◆ ellenőrizhető

Event

- Nyelvbe épített event modell, a delegátokra épül

- Eseményt deklarálni és hívni csak a tulajdonos (publikáló) osztályból lehet

```
public delegate void ClickHandler(  
    object sender, EventArgs e);
```

```
event ClickHandler OnClicked;
```

```
...
```

```
OnClicked(this, PointerLocation); //elsüti
```

- Eseményre beregisztrálás kezelőfüggvény hozzáadásával az előfizető objektumnál:

```
Button.OnClicked += MyClickHandler;
```

```
// (a MyClickHandle az előfizető egy művelete)
```

- A management C#-ban történik

Event példa – publikáló

```
public class MyClass
{
    public delegate void LogHandler( string msg );
    public event LogHandler Log;
    public void Process()
    {
        if( Log != null )
            Log( "Process() begin ..." );
        ...
        if( Log != null )
            Log( "Process() end" );
    }
}
```

Event példa - előfizető

```
class Test
{
    static void Logger( string s ) {
        Console.WriteLine( s );
    }

    public static void Main() {
        MyClass instance = new MyClass();

        instance.Log += new MyClass.LogHandler( Logger );
        instance.Process();
    }
}
```

Struktúra

- **Adat és kód csoportosítva**
 - ◆ Hasonló az osztályhoz, de:
 - nincs öröklődés, csak interfész implementálás
 - *mindig érték szerint átadva*
 - ◆ Osztály vs. Struktúra
 - struktúra ⇒ Könnyű adat kontainer, érték típus
 - osztály ⇒ Gazdag objektum referenciaikkal, ref. típus
- **C++-szal ellentétben *struct* nem egy olyan *class* hogy minden *public***

```
struct Point
{
    double X;
    double Y;
    void MoveBy(double dX, double dY)
    { X+=dX; Y+=dY; }
}
```

Tömbök

- Nulla alapú, erősen típusos, a .NET System.Array osztályán alapul
- Deklaráláskor típus és dimenziók, de nincs korlát
 - ◆ `int[] EgyDim;`
 - ◆ `int[,] KetDim;`
 - ◆ `int [][] Beagyazott;`
- Létrehozás `new`-val (korlátokkal vagy inicializálással)
 - ◆ `EgyDim = new int[20];`
 - ◆ `KetDim = new int[,] { {1,2,3}, {4,5,6} };`
 - `KetDim[0,0] == 1 !`
 - ◆ `Beagyazott = new int[2][];`
 - ◆ `Beagyazott[0] = new int[]{1,2,3};`
 - ◆ `Beagyazott[1] = new int[]{4,5};`
 - `Beagyazott[0][2] == 3 !`

Tömbök

- Érték típusú elemekkel (struktúrák! is) – egyben lefoglalja

```
Person [] persons = new Person[20];
```

- Referencia típusú elemekkel

```
Person [] persons = new Person[20];
```

```
for (int i = 0; i < persons.Length; i++ )  
    persons[i] = new Person();
```

Verziókezelés

- **Nem csak virtuális metódusok vannak**
- **Az metódusoknál explicit jelölni kell az elvárt működést**
 - ◆ ellentétben a Java-val vagy C++-szal
- **Ütközés esetén elvárt default működés (de a fordító figyelmeztetést generál)**

Metódus verzionálás Java-ban

```
class Base // v3.0
{
    public int Foo()
    {
        return Database.Log("Base.Foo");
    }
}
```

```
class Derived extends Base // v1.0
{
    public void Foo()
    {
        System.out.println("Derived.Foo");
    }
}
```

Metódus verzionálás C#-ban

```
class Base // v3.0
{
    public virtual int Foo()
    {
        return Database.Log("Base.Foo");
    }
}
```

```
class Derived : Base // v3.0
{
    public override void Foo()
    {
        Base.Foo();
        Console.WriteLine("Derived.Foo");
    }
}
```

Virtual

- Futási időben dől el, hogy melyik függvény (vagy tulajdonság) hajtódik végre – a származtatási láncban az utolsó
- Az **override** kulcsszóval lehet felüldefiniálni a virtuális függvényeket
- Nem használható az **static**, **abstract** és **override** kulcsszavakkal együtt !

Override

- Az **override** megváltoztatja az eredeti virtuális vagy absztrakt fv (vagy tulajdonság) implementációját
- A felüldefiniált fv-nek virtuálisnak vagy absztraktnak kell lennie
- Az **override** nem változtathatja meg az elem elérési szintjét (public, protected, .)
- Nem használható a *new*, *static*, *virtual*, *abstract* kulcsszavak együtt !

A new módosítószó

- Teljesen új függvényt hoz létre, megszakítja a virtuális fv-láncot
- Konstanst, mezőt, tulajdonságot, függvényt vagy típust lehet újként definiálni (azaz a régit elfedni)
- Nem használható az *override*-dal együtt

Összefoglalva

A virtuálissal lehet megcsinálni, hogy mindenkor a származott osztály függvénye hívódjon meg, még ha az alaposztályból hívsz, akkor is. A *virtuális* függvényt az *override*-dal lehet felüldefiniálni.

Egyébként új függvény létrehozására (a szignatúra csak véletlenül azonos) a *new* használandó (beágyazott osztályra, stb.-re is).

Lehet *new virtual*: ilyenkor egy új lánc indul, az eddigi osztályok az előző virtuális fv-t hívják meg, az ez után következők viszont az ebben a láncban lévő utolsót.

Hozzáférési jogok

- **Átveszi a C++ modellt**
 - ◆ **public** ⇒ mindenki hívhatja/elérheti
 - ◆ **protected** ⇒ csak tagok hívhatják/érhetik el
 - ◆ **private** ⇒ csak pontosan ennek a tagjai
- **Kibővíti a C++ modellt**
 - ◆ **sealed** ⇒ Nem lehet belőle származtatni
 - ◆ **internal** ⇒ publikus elérés csak Assembly-n belül
 - ◆ **protected internal** ⇒ védett Assembly-n belül és a származott típusokra

Cast

- **Cast – kivételt dob, ha nem lehet**

```
void f(Shape s) {  
    ((shape)s).Radius = 10;  
}
```

- **as -null-t ad vissza, ha nem lehet**

```
void f(Shape s) {  
    (shape as Circle).Radius = 10;  
}
```

- **is**

Attribútumok I.

- **.NET terminológiában: Nem az osztály tagja (OO elnevezés) – az a mező**
- **Metainformáció rendelhető minden nyelvi elemhez**
 - ◆ osztályokhoz, metódusokhoz, tulajdonságokhoz, paraméterekhez, mezőkhöz, szerelvényhez, stb.
- **Ezek az információk megváltoztathatják a fordító viselkedését (beépített attribútumok) VAGY**
- **Futási időben lekérdezhetőek**
 - ◆ Használja őket a keretrendszer
 - ◆ Használhatók saját kódban
- **Saját attribútumok alkothatók**
 - ◆ Saját kódban le kell őket kérdezni, egyébként nem jelentenek semmit !

Attribútumok példa (C#)

```
[HelpUrl("http://SomeUrl/Docs/SomeClass")]
[Serializable]
class SomeClass
{
    [WebMethod]
    [MethodImpl(MethodImplOptions.Synchronized)]
    void SetupCustomers() { ... }

    [NonSerialized]
    int n;

    string Test([SomeAttr] string param1) {...}
}
```

Attribútumok II.

- **Natív kóddal való együttműködés testreszabásához**
 - ◆ layout, string, ...
- **Perzisztens osztályok létrehozásához**
 - ◆ Beépített sorosítás
 - ◆ XML perzisztencia testreszabásához
- **Metódusszintű deklaratív jogosultság ellenőrzés**
 - ◆ Kész alkalmazás utólagos ellenőrzése, jogosultságok feltérképezése
 - ◆ Hívó szerelvények ellenőrzése, integritás biztosítása
- **Tranzakcióban résztvevő osztályok megjelelöléséhez**
- **Web szolgáltatások készítéséhez**

Konstans vagy csak olvasható

Konstans

Fordítás-idejű kiértékelés
Mindig statikus

Csak olvasható

Futásidőjű kiértékelés
Statikus vagy tag változók

```
class Math
{
    public const double Pi = 3.14;
}

class Color
{
    public static readonly Color Red    = new Color(...);
    public static readonly Color Blue   = new Color(...);
    public static readonly Color Green = new Color(...);
}
```

Statikus konstruktorok

- Globális változók inicializálása
- Az első példány létrejötte illetve az első statikus függvényhívás, változóelérés előtt meghívódik

```
class CharInfo
{
    static bool[] isAlpha;
    static int Min = -9999;
    static int Max = 9999;

    static CharInfo() {
        isAlpha = new bool[256];
        for (int c = 'A'; c <= 'Z'; c++) isAlpha[c] = true;
        for (int c = 'a'; c <= 'z'; c++) isAlpha[c] = true;
    }
}
```

Destruktorkok – későbbi órán részletesen

- **Mark-and-sweep szemétgyűjtés – nemdeterminisztikus destruktorkok**
 - ◆ a végrehajtás ideje nem ismert
 - ◆ a sorrend nem ismert
 - ◆ a szál (!!) nem ismert
- **A destrukturor csak a külső hivatkozásait engedheti el**
- **Destruktorkokkal rendelkező objektumok tipikusan implementálják az IDisposable interfészt**

Destruktorok – későbbi órán részletesen

- Az **Object.Finalize** nem elérhető C#-ból

```
public class Resource: IDisposable
{
    ~Resource() {...}
}
```

```
public class Resource: IDisposable
{
    protected override void Finalize()
    {
        try {
            ...
        }
        finally {
            base.Finalize();
        }
    }
}
```

Destruktörök

```
public class Resource: IDisposable
{
    public void Dispose() {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing) {
        if (disposing) {
            // Dispose dependent objects
        }
        // Free unmanaged resources
    }

    ~Resource() {
        Dispose(false);
    }
}
```

Pointerek ? – ref és out paraméterek

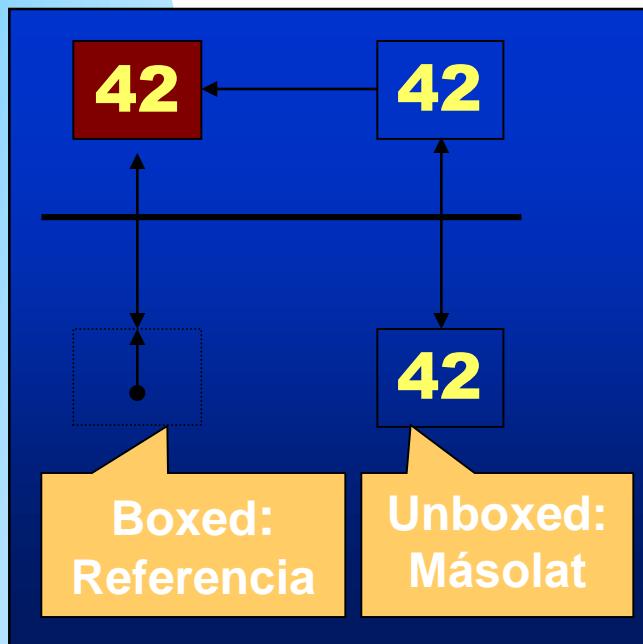
- C# támogatást ad:
 - ◆ Beépített **string** típus
 - ◆ Gazdag kollekció modell
 - ◆ Referencia szerinti átadás ref kulcsszóval
void increment(ref int value, int by)
...
int i = 1; int j;
increment(ref i, j);
 - ◆ Kimeneti paraméterek az out kulcsszóval
bool add(int a, int b, out int c)
 - ◆ A ref változót inicializálni kell, az out-ot nem
- A legtöbb C++ pointer használat szükségtelen

ref és out paraméterek példa

```
void increase( ref int a, int by )  
{ a += by; }  
  
void decrease( int a, int by, int ret )  
{ ret = a - by; }  
  
void decrease2( int a, int by, out int ret )  
{ ret = a - by; }  
  
...  
  
int a = 3, by = 2;  
int r; // nincs inicializalva  
  
increase( a, by );  
decrease( a, by, r );           // nem fordul !!  
decrease2( a, by, r );
```

Boxing és Unboxing (be- és kidobozolás)

- Érték szerinti típusok boxolhatóak és boxolás után unboxolhatók
- "Boxing" lehetővé teszi az érték típusú objektumok referenciakénti átadását
- Alapja az, hogy minden alaptípus objektum is
- Gondolat: dobozba dobjuk az értéket és referenciáljuk



`double value;`

```
// Boxing  
object Boxedvalue = value;  
// Unboxing  
value = (double)Boxedvalue;
```

Structs vs. Classes

- **A fő kérdés: struct legyen valami vagy class (a többi úgyis adott).**
 - ◆ Ha csak egyszerű adattároló: struct.
 - ◆ Ha polimorf kell legyen, akkor csak class lehet.
- **Ne feledjük –a struct érték típus**
 - ◆ Ha átadunk egy példányt függvénynek, vagy betesszük egy collection-be (pl. ArrayList) akkor másolat készül (és a fv. ezt változtatja meg)
 - ◆ **Lásd még tömbök**

Kivételkezelés

- **(try) próbáljuk meg futtatni ezt ...**
- **...ha hiba van kapjuk (catch) el amit le tudunk kezelní...**
- **... (finally) végül takarítsunk minden esetben**
- **Exception osztály és leszármazottai. Tulajdonságok:**
 - ◆ StackTrace (string)
 - ◆ InnerException
 - ◆ Message
 - ◆ HelpLink
 - ◆ TargetSite – a dobó függvény
 - ◆ Source – forrás alkalmazás/szerelvény

Névterek

- **Minden definíciónak namespace-ben kell lennie**
 - ◆ Elkerüli a névütközéseket
 - ◆ Egyszerűbbé teszi az API-k átlátását
- **Egymásba ágyazható (ajánlott)**
- **Osztályok, típusok szemantika elvű csoportosítása**
- **A namespace kulcsszóval deklarálhatóak**
- **Referálás a using kulcsszóval**

Típusok teljes neve

- A típus teljes névéhez mindenkor hozzá tartozik a névtér
 - ◆ NEM: string
 - ◆ HANEM: System.String
- A névtér független a tároló szerelvénytől
- A típus teljes neve tartalmazza a szerelvény azonosítását is (több szinten, biztonság)

*System.String, mscorelib, Version=1.0.5000.0,
Culture=neutral,
PublicKeyToken=b77a5c561934e089*

Beépített kollekciók, foreach

- **Egyszerű támogatás kollekciók iterálására**
 - ◆ használható tömbökre és más kollekciókra
- **Használható saját egyedi osztálynál**
 - ◆ Ez implementálja az **IEnumerable**-t (**GetEnumerator()**)
 - ◆ A **GetEnumerator()** visszaad egy objektumot, ami implementálja az **IEnumerator**-t
 - (Műveletek: Reset, method MoveNext, property Current)

```
Point[] Points = GetPoints();
foreach( Point p in Points )
{
    MyPen.MoveTo( p.x,p.y );
}
```

Formázás

```
c.ToString("C") - 100 Ft v. $100
c.ToString("C", new CultureInfo("hu-HU")) ;  
  
string myName = "Fred";
String.Format("Name = {0}, hours = {1:hh}, minutes
= {1:mm}", myName, DateTime.Now);  
  
// Consol.WriteLine, TextWriter.WriteLine is
támogat formázást.  
  
Color[] colors =
(Color[])Enum.GetValues(typeof(Color));
foreach( Color j in colors )
    Console.WriteLine( "{0:G} = {0:D}", j );
```

Basic IO

- **System.IO névtér**
- **File, Directory, DirectoryInfo, FileInfo, FileSystemInfo, Path**
- **System.IO.Stream**
 - ◆ System.IO.FileStream , ...
- **BinaryWriter, BinaryReader**
 - ◆ A stream, amin dolgozik, binárisan reprezentált
- **TextWriter (StreamWriter, StringWriter) és
TextReader (StreamReader, StringReader)**
 - ◆ Amin dolgozik, az szövegesen reprezentált. Fizikailag:
 - String objektum
 - Stream objektum

XML megjegyzések

```
/// <summary>
///     Returns the attribute with the given name and
///     namespace</summary>
/// <param name="name">
///     The name of the attribute</param>
/// <param name="ns">
///     The namespace of the attribute, or null if
///     the attribute has no namespace</param>
/// <return>
///     The attribute value, or null if the attribute
///     does not exist</return>
/// <seealso cref="GetAttr(string)" />
/// <exceptions>ArgumentNullException</exceptions>
public string GetAttr(string name, string ns) {
    ...
}
```

- **XML séma**
- **Fordító és fejlesztőeszköz támogatás**

Megjegyzések XML-ben

- Konzisztens út a kódból való dokumentáció készítésére
- "///" megjegyzések exportálása
- Fordítás közben generálja a dokumentációt (/doc)
- Előredefiniált sémákkal rendelkezik

Különbségek: C++ és C#

- **A C# nagyon hasonlít a C/C++-ra**
- **Megszüntet sok hibalehetőséget:**
 - ◆ szigorúbb típusellenőrzés, kevés type-cast
 - ◆ nincs „átesés” a **switch**-ben
 - ◆ a boolean kifejezések erősen típusosak
 - ◆ jobb elérés védelem, nincs trükközés a headerekkel
 - ◆ nincs pointer
 - ◆ nem kell üldözni a memória szivárgást

Irodalom

- **Útmutatók:**

- ◆ [https://msdn.microsoft.com/en-us/library/aa288436\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/aa288436(v=vs.71).aspx)

Win32 alkalmazások architektúrája és eseményvezérelt programozás Win32 környezetben

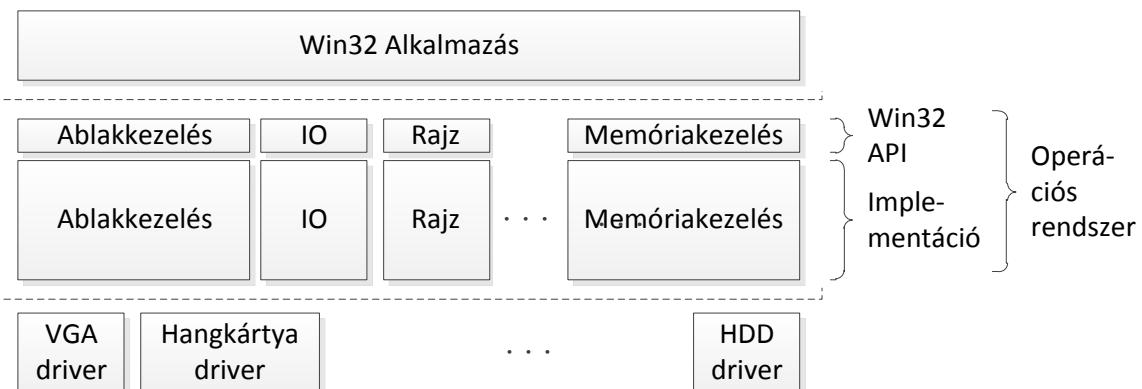
Szoftvertechnikák jegyzet

BME-AAIT Dr Charaf Hassan, Benedek Zoltán

Cél

Megnézzük, hogy natív módon, felügyelt környezetek nélkül hogyan tudunk alkalmazást fejleszteni, de most már ablakos felhasználói felülettel rendelkezőt. A felügyelt környezetek is erre építenek nagy részben.

OS felépítése, natív alkalmazások



Kiegészítő magyarázat



API (Application Programming Interface)

API alatt a Windows alatt használt és a fent említett DLL-ekben definiált főleg függvények, (változótípusok és konstansok) összességét értjük. Ezek az elemek nem nyelvfüggőek. Bármilyen Windows alatt használt programozási nyelvben alkalmazhatók (C, C++, MS-ACCESS, VISUAL BASIC, FOXPRO, Delphi, stb.).

Változótípusok

Természetesen minden nyelvnek vannak saját változótípusai. Ezeken a "hagyományos" típusokon kívül vannak olyan típusok, amelyek Windows specifikusak. A C nyelv esetén ezek a <windows.h> fájlban vannak definiálva. Ezek közül néhány: LONG, WORD, DWORD, LPSTR, HWND, HANDLE, LRESULT, stb. Fontos megértenünk, hogy ezek a típusok egy adott nyelv esetében mindenkorán az adott nyelv egy típusára képződnek le. Pl. a C illetve C++ nyelv esetében:

```
typedef unsigned int UINT; // insigned int a C nyelv beépített típusa  
typedef long LONG; // long a C nyelv beépített típusa
```

```
typedef UINT WPARAM;
typedef LONG LPARAM;
```

Számos struktúra típus definiált az Windows API-ban, ezek közül néhányat említünk:

```
typedef struct tagPOINT{ int x; int y;}POINT
typedef struct tagMSG { HWND hwnd; UINT message; WPARAM wParam;
                        LPARAM lParam; DWORD time; POINT pt;}MSG;
typedef struct tagRECT {int left; int top; int right; int
                        bottom;}RECT, stb...
```

Magyar jelölési rendszer

A Windows programozás során a változó neveket - Simonyi Károly nyomán - úgy szokás felvenni, hogy a név kezdőbetű utaljanak a változó típusára. Pl.:

b : char	n :int
p : 16 bites pointer	l :long
lp: 32 bites pointer	

Ezt a konvenciót nem kötelező követni, de célszerű ismerni, mert a WinAPI dokumentációja is ezt használja.

Konstansok, attribútumok

Több 1000 előre definiált konstans van az API-ban. Ezeknek az a jellegzetessége, hogy az elnevezésük prefixumokkal (2, 3 nagybetű) kezdődik. A prefixum utal a konstans szerepkörére. A prefixum után egy aláhúzás jel következik és utána jön maga a konstans nagybetűvel. Néhány példa illusztrációként:

CS: (<u>Class Style</u>) ablakosztály stílus	CS_HREDRAW
IDI: (<u>IDentifier Icon</u>) ikon azonosító	IDI_APPLICATION
IDC: (<u>IDentifier Cursor</u>) Kurzor azonosító	IDC_WAIT
WS: (<u>Window Style</u>) ablak típus	WS_ICONIC
CW: (<u>Create Window</u>) ablak létrehozás	CW_DEFAULT
WM: (<u>Windows Message</u>) Windows üzenet	WM_PAINT
DT: (<u>Display Text</u>) szöveg megjelenítés	DT_CENTER
BM: (<u>Box Message</u>) Check- vagy RadioBox üzenete	BM_GETCHECK
CB: (<u>Combo Box</u>) kombó doboz	CB_GETCURSEL
LB: (<u>List Box</u>) lista doboz	LB_SETCURSEL
EM: (<u>Editor Message</u>) editor üzenet	EM_LIMITTEXT
CF: (<u>Clipboard Format</u>) vágólap formátuma	CF_BITMAP
ERR: (<u>ERRor</u>) hibakódok, stb...	ERR_CREATEDLG

Ezek a konstansok általában egy adott számot jelölnek.

Függvények

A teljes Win32 API sok ezer API függvényt tartalmaz. A függvények nevei általában több angol szóból állnak. A szavak első betűje minden nagybetű pl. SetWindowText(...). Természetesen a saját függvényeinket úgy nevezzük, ahogy akarjuk. Figyelembe kell venni, hogy a C nyelv különbséget tesz a nagy és a kis betű között (case sensitive). Ha ugyanazt az API függvényt akarjuk meghívni egy másik programozási nyelven pl. a BORLAND PASCAL-ban vagy VISUAL BASIC-ben vagy MS-ACCESS-ben, akkor nem kell különbséget tenni a kis és a nagy betű között, de érdemes (esztétikailag) úgy írni a függvény nevét, ahogy a C-ben írjuk és ahogy szerepelt a dokumentációban.

Az API függvények **WINAPI** (régebben **FAR PASCAL**) „típusúak”. Ez a függvények hívási konvencióját határozza meg: a függvényhívás a Pascal nyelv konvenciója szerint történik. Mint ismeretes a Pascal konvenció azt jelenti, hogy a függvény paramétereit a deklaráció sorrend alapján jobbról balra kerülnek a stack-be és a stack kiürítése (takarítása) a hívott függvény feladata.

Vannak olyan kitüntetett szerepű függvények, amelyeket mi írunk meg, viszont az operációs rendszer hív meg (jelen esetben a Windows!). Ezeket a függvényeket **CALLBACK** függvényeknek nevezzük. A CALLBACK típusú függvények hívása is a Pascal hívási konvenció szerint történik.

A WINAPI és a CALLBACK szavak a fordítónak (compiler) szólnak és a függvény neve előtt kell őket írni, pl:

```
LONG CALLBACK WndProc (HWND, UINT, UINT, LONG);
```

Objektumok és leírók (HANDLE-k)

Amikor a Windows-ban létrehozunk valamilyen objektumot (itt most ne az objektum-orientált nyelvek objektumaira gondoljunk), akkor általában azt egy leíró (handle) fogja azonosítani. Például amikor létrehozunk egy ablakot, akkor az ablakot létrehozó függvény az új ablaknak a leírójával (HWND) tér vissza. Ezt a leíró azonosítja az újonnan létrehozott ablakot, ezt a leíró kell az egyéb ablakkezelő függvényeknek átadni. Egy leíró általában egy 32 bites érték, típusa pedig HANDLE vagy valamilyen más H betűvel kezdődő típus. Ilyen objektumok például a következők:

Objektum	Leíró típus
ablak	HWND
processz	HANDLE
szál	HANDLE
eszközkapcsolat rajzoláshoz és nyomtatáshoz	HDC
betűtípus	HFONT
fájl	HANDLE

Windows verziók és a Win32 API

Az egyes Windows verziók – pl. Win95 és Win2000 - architektúrája, belső működése alapjaiban különböző. Ennek ellenére mivel ugyanazon a Win32 API-n keresztül programozzuk őket, a két platformon futó programok elkészítésében általában nincs különbség. Ennek ellenére előfordulhat, hogy bizonyok API függvények csak az egyik illetve másik platformon

értelmezettek, vagy működésük a két platformon némi képp eltér. Ezen felül ugyanaz - a már lefordított program a legtöbb esetben egyaránt futtatható minden platformon.

Eseményvezérelt programozás

Eseményvezérelt programok

A hagyományos programozás során minden program hívja az operációs rendszert (pl. billentyűzet vizsgálata, olvasás a standard inputról/írás a standard outputra). A Windows programok működése a "hagyományos" programoktól alapvetően eltér. Windows alatt futó programok esetében ennek pont a fordította történik: a programunk várakozik, az operációs rendszer hívja a programunkat (illetve annak egy függvényét), ha valamilyen esemény bekövetkezik a rendszerben. Milyen események lehetségesek: a felhasználó lenyomott egy billentyűt, kattintott az egérrel, kiválasztott egy menüelemet, stb. Így a Windows alatt futó programok futási menetét a felhasználói események határozzák meg, program ezekre az eseményekre reagál. Így programunk működése némi képp az interrupt vezérelt programokéra hasonlítható.

Ennek megfelelően a Windows rendszer üzenet alapú. Amikor egy esemény történik pl.: egér mozgatás, gomb kattintás, billentyűzeten lenyomtak egy gombot, akkor egy üzenetet generál a Windows, bár maga az applikáció is generálhat üzeneteket. Az applikáció reagálása ezekre az eseményekre – vagyis az üzenetek kezelése – határozza meg a program menetét illetve viselkedését. Az applikációhoz érkező üzenetek számos információval rendelkeznek az eseményről: típusa, keletkezési ideje, és melyik ablaknak szól az üzenet.

Üzenetek

A felhasználói eseményeket (egér, billentyűzet, időzítés, menü stb..) a Windows saját egységes üzenet formátumma konvertálja. Az üzenet a következőket tartalmazza: a címzett ablak leírása (HWND), az üzenet tárgya/tartalma (UINT típusú szám), az üzenet paraméterei (WORD wParam, LONG lParam), az üzenet keltének időpontja és az üzenet keltésekor érvényes kurzor pozíció. Az üzenetek MSG típusú struktúrában tárolhatók:

```
typedef struct tagMSG
{
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
}MSG;
```

Az üzenet címzettje – hwnd

Mint látható, minden üzenetnek van egy címzett vagy cél ablaka. Egyszerre több programot lehet indítani a Windows alatt. Aktív applikáció alatt azt az applikációt értjük, amely éppen elfogadja a felhasználói eseményeket. Egy applikáción belül aktív ablakról is beszélhetünk. Az egér és billentyűzet üzenetek címzettje az az ablak, amelyik az esemény bekövetkezésekor aktív volt (vagyis amelyik ablakon a felhasználó kattintott az egérrel illetve amelyik ablakba a felhasználó „gépel”). Az üzenetet végső soron a címzett ablak (illetve annak ablakkezelő függvénye – lásd később) kapja meg.

Az üzenet típusa - message

Egy adott üzenet típust egy egyedi szám azonosít. A könnyebb kezelhetőség érdekében az egyes üzenet típusok a windows.h -ban konstansként vannak definiálva, így egy semmitmondó szám helyett egy sokkal beszédesebb névvel is tudunk hivatkozni rájuk (pl. 0x0100 helyett WM_KEYDOWN). Például billentyű lenyomását illetve a bal egérgomb lenyomását jelentő számaazonosítóra a konstansok definiálása:

```
#define WM_KEYDOWN          0x0100
#define WM_LBUTTONDOWN        0x0201
```

Azt szoktuk mondani, hogy egy ablak kap egy WM_KEYDOWN vagy egy WM_LBUTTONDOWN típusú üzenetet.

Az üzenet paraméterei – wParam és lParam

Minden üzenethez tartozik két, az üzenet típusától függő paraméter: egy WPARAM típusú és egy LPARAM típusú. Valójában minkét típus egy 32 bites egész értéket (integer-t) jelent. Például egy WM_KEYDOWN típusú üzenetnél ezek a paraméterek tartalmazzák a lenyomott billentyű kódját és más jellemzőit. A wParam és az lParam 32 bites értékek, több üzenet típus esetében összetett információt tartalmaznak. Ilyenkor segíthet az alábbi két makró:

LOWORD (...) - a paraméterben megadott 32 bites érték alsó szavát (alsó 16 bit) adja vissza
HIWORD (...) - a paraméterben megadott 32 bites érték felső szavát (felső 16 bit) adja vissza

Fontosabb üzenettípusok

- Egér események – egér mozgatásakor, egérrel való kattintáskor
- Billentyűzet események – billentyű lenyomásakor
- Időzítő események
- Az operációs rendszer működésével kapcsolatos események. Pl.:
 - WM_CREATE – az ablak létrejöttekor kapja meg az adott ablak
 - WM_DESTROY – az ablak megszüntetésekor kapja meg az adott ablak
 - WM_SIZE – ha az ablak mérete megváltozott, mert pl. a felhasználó átméretezte az ablakot
 - WM_MOVE – az ablak pozíciója megváltozott
 - WM_PAINT – az ablakon érvénytelen területek keletkeztek, az ablak újra kell rajzolja magát
 - WM_VSCROLL és WM_HSCROLL – a görgetősávval kapcsolatos esemény következett be
- Parancs (command) üzenetek: WM_COMMAND. Ez egy sokcélú üzenet típus, pl. akkor keletkezik, ha:
 - a felhasználó kiválaszt egy menüelemet
 - a felhasználó kattint egy toolbar gombon,
 - a felhasználó megnyom egy gyorsítóbillentyűt,
 - ha egy gyerekablakkal történik valami, akkor általában WM_COMMAND-ot küld a szülő ablaknak és az üzenet paramétereiben küldi el az esemény leírásátTermészetesen a WPARAM és LPARAM értelmezése az egyes esetekben más és más.
- A felhasználó által definiált üzenetek (WM_USER + N és WM_APP + N, ahol N egy szám)

Ablakkezelő függvény (Window Procedure)

A Windows-ban szinte minden olyan vizuális elem ablak (window), amelyik valamelyen interakcióra képes a felhasználóval. Így a közönséges ablak mellett ablakok a nyomó- és kiválasztógombok, szövegmezők, dialógusablakok, stb. Az egyes ablakok csak azért viselkednek másként (vagyis egy nyomógomb azért más mint egy szövegmező), mert ugyanazokra az üzenetekre másképpen reagálnak.

A Windows-ban minden ablakhoz tartozik egy **ablagkezelő függvény** (CALLBACK típusú függvény), amely gondoskodik az ablakhoz érkező események kezeléséről. Mint már volt róla szó a Windows értesíti a programunkat arról, ha valamilyen esemény következik be, erre az alkalmazás valamilyen módon reagálhat (ettől eseményvezérelt program). Most már azt is tudjuk milyen módon értesíti a programunkat: **meghívja a célablak ablakkezelő függvényét és paraméterként átadja neki az üzenetet**. Ennél fogva ezen függvényen keresztül kommunikál a Windows az alkalmazásunkkal. A függvényen belül egy switch - case szerkezetben megvizsgáljuk milyen üzenet érkezett és az érkezett üzenettől függően csinálunk valamit (pl. rajzolunk, stb).

Egy programhoz több ablak tartozhat és legalább egy mindig tartozik. Ebből adódik, hogy egy Windows programban legalább egy ablakkezelő függvény van.

Üzenetek kézbesítése a megfelelő ablakkezelő függvényhez

Üzeneteknek a célablakhoz (illetve annak ablakkezelő függvényéhez) való eljuttatásának kétféle módja lehet:

a, **Queued**: az üzenet bekerül az üzenetsorba és onnan továbbítódik az ablakkezelő függvények. Ezt a fajta üzenet kézbesítést üzenet feladásnak (post a message) nevezzük

b, **Not-queued**: az üzenetsor kikerülésével közvetlenül az ablakkezelő függvény kapja meg az üzenetet. Ezt a fajta üzenet kézbesítést üzenet küldésnek (send a message) nevezzük

a, Queued üzenetek és üzenet sorok

Minden thread rendelkezik egy üzenet sorral. Ebbe a sorba kerülnek a szálhoz érkező üzenetek és onnan továbbítónak a megfelelő ablakkezelő függvényhez. Ezeknek az üzeneteknek a feldolgozása **aszinkron** módon történik. Az üzenetsor a postaládához hasonlít: a küldő nem vár az üzenet elolvasására, csak küldi a levelet és abban bízik, hogy a fogadó valamikor el fogja olvasni azt. minden szál maga dolgozza fel a saját üzeneteit – vagyis továbbítja azokat a megfelelő ablak ablakkezelő függvényének. Az üzenet feldolgozás FIFO módon történik.

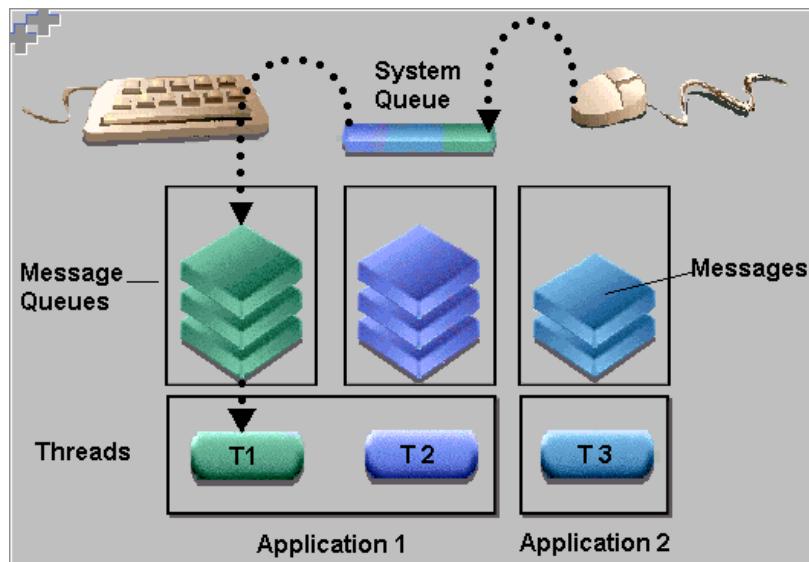
Az billentyűzet és egér üzenetei minden aszinkron módon dolgozódnak fel. Ezeket az eseményeket az üzenetté való konvertálása után a rendszer egy rendszer várakozási sorba (system queue) helyezi el. **Minden szálhoz (GUI thread) tartozik egy üzenet sor** (message queue). A system queue -ba érkező üzeneteket az operációs rendszer továbbítja az applikációk szálainak üzenet soraiba.

Megjegyzés

Amikor az applikáció elindul, vagyis processz lesz belőle, minden elindul egy szál (ettől fog a processz „futni”). Az egyszerűbb applikációk általában egy szalon futnak, így azt is szokták mondani, hogy üzenetsora applikációnak van. Ez természetesen csak addig igaz, amíg nem indítunk más szálakat.

Megjegyzés

Üzenetsora valójában nincs minden szálnak, hanem csak akkor rendelődik üzenetsor egy adott szálhoz, amikor az a szál az első grafikus vagy ablakkezelő műveletet végrehatja. Ez az erőforrásokkal való takarékosság miatt van így, hiszen lehet, hogy egy szál (worker thread) soha nem hoz létre ablakot és így nem kell fogadjon üzeneteket. Ebben az esetben az üzenetsor létrehozásának nincs értelme.



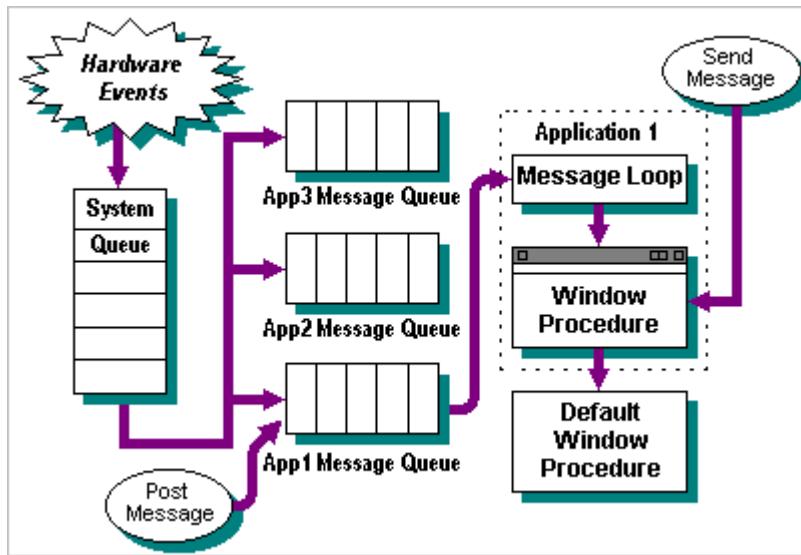
b, Not-queued üzenetek

Nem minden üzenet kerül be a sorba. Lehetőség van arra, hogy szinkron módon is küldjünk üzenetet egy ablaknak. Ebben az esetben az üzenet nem kerül be a szál üzenetsorába, hanem közvetlenül az ablak ablakkezelő függvénye hívódik meg. Ebben az esetben, a küldő addig vár, amíg nem tér vissza a függvény, vagyis az üzenetet a célablak fel nem dolgozza.

Üzenetkezelő ciklus (Message Loop)

Az applikáció (szál) maga gondoskodik az üzenet kiszedéséről és feldolgozásáról: ezt végezi az üzenetkezelő ciklus (message loop).

Minden applikáció rendelkezik legalább egy üzenetsorral (mivel legalább egy szál tartozik hozzá). Kérdés, hogyan jutnak el az üzenetek az üzenetsorból a megfelelő ablakkezelő függvényhez. Azt gondolnánk, hogy az operációs rendszer ezt teljesen elrejti és a programozó feladata pusztán az eseményekre való reagálás azáltal, hogy megírja ablak(ok) ablakkezelő függvényét. Ez azonban nem így van. minden applikációnak (minden GUI thread-nek) tartalmaznia kell egy **üzenetkezelő ciklust**, ami kiszedi az üzeneteket az üzenetsorából és azt továbbítja a megfelelő ablakkezelő függvénynek.



Üzenet küldés

Üzeneteket mi is küldhetünk ablakoknak, mégpedig kétféle módon: a várakozási soron keresztül (queued) vagy ennek elkerülésével (not-queued).

Queued üzenetet a **PostMessage** függvényteljesítményével küldhetünk, nonqueued-ot a **SendMessage** függvény segítségével.

```
PostMessage(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
SendMessage(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
```

A hWnd a célablak leírója, a message az üzenet típusa, wParam és lParam pedig az üzenet paraméterei.

A PostMessage beteszí az üzenetet a célablakot kezelő szál üzenet sorába és azonnal visszatér. A SendMessage közvetlen a célablak ablakkezelő függvényét hívja meg és addig nem tér vissza, amíg a célablak az üzenetet fel nem dolgozta. Az üzenetek nem hasonlíthatók a megszakításokhoz. Az egyik üzenet feldolgozását egy másik üzenet nem szakíthatja meg. Ennek elkerülésére használhatjuk a SendMessage-et úgy, hogy az üzenet feldolgozása közben egy másik üzenetet küldünk.

Üzenetek alapértelmezett feldolgozása

Ablakkezelő függvényben vannak olyan üzenetek, amelyeket a program kezel és vannak olyanok, amelyeket az alapértelmezett (default) ablakkezelő függvénynek juttat el. A **DefWindowProc** függvény szolgál az üzenet default ablakkezelő függvényhez való továbbítására. Olyan üzenetek szokás továbbítani, melyek a program szempontjából érdektelenek és az alapértelmezett módon akarjuk lekezelni. Ilyen pl. az ablak minimalizálás, maximalizálás, a menük megjelenítése, stb.

A program szerkezete

Már volt róla szó, hogy minden üzenetnek van egy címzett ablaka. Amikor egy windows applikáció indul, a program egy adott pontban kezdi el a futását: ez a belépési pont (entry point) a **WinMain** függvény. Az applikáció létrehoz egy vagy több ablakot. minden ablak tartalmaz egy ablakkezelő függvényt, amely gondoskodik az ablak üzeneteinek feldolgozásáról. A program tartalmaz egy üzenetkezelő ciklust. Ez a ciklus kiolvassa az üzeneteket és visszaadja a Windows-nak, hogy meghívja az ablakkezelő függvényt, ami egy **CALLBACK** függvény, tehát nem a mi programunk hívja, hanem maga a Windows.

Egy tipikus Windows program a következő részeiből áll:

1. Inicializálás: ebben a részben történik az ablak osztály regisztrálása a rendszerben és a változók inicializálása.
2. A főablak létrehozása.
3. Az üzenetkezelő ciklus (kiolvassa a várakozási sorból az üzeneteket és továbbítja a Windowsnak, hogy ezekkel a paraméterekkel hívja meg az ablakkezelő függvényt).
4. A főablak kezelő függvénye (Window procedure): az üzenetekre való reagálás
5. A kilépés előtti takarítás, kilépés.

Egy tipikus program váza:

```
#include <windows.h>
#include <másfájlok.h>
#include "myfiles.h"

// globális változók deklarálása

WinMain(.....)
{
    //Lokális változók deklarálása
    InitProgram();

    // ablakosztaly regisztrálása
    WNDCLASSEX wcex;
    ...
    wcex.lpfnWndProc = WndProc;
    RegisterClassEx(&wcex);

    // ablak létrehozása
    hWnd=CreateWindow(....)

    // ablak megjelenítése (maximized, minimized, default)
    ShowWindow(...);
    UpdateWindow(hWnd);

    // Üzenetkezelő ciklus
    While(GetMessage(&msg, NULL, NULL, NULL))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```

```
// Ablakkezelő függvény
LONG CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    // lokális változók

    // egy switch - case szerkezetben megvizsgáljuk milyen üzenetet kapott a
    // a függvény paraméterként és az üzenettől függően csinálunk valamit
    Switch (msg)
    {
        case WM_CREATE:
            .... return 0;
        case WM_PAINT:
            RajzoljValamit();
            return 0;
        case WM_COMMAND:
            switch (wParam)
            {
                case IDM_FILENO: // ez pl. lehet egy menü elem azonosító
                    CsinaljValamit();
                    break;
            }
            return 0;
        case WM_DESTROY:
            memóriafelszabaditas() // pl. memória felszabadítás
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hwnd, msg, wParam, lParam);
}
```

Mint tudjuk a C nyelvben a main() függvény a belépési pontja a programnak. A Windows alatt a main függvény helyet a **WinMain**-t kell használni. Az API elemek a windows.h-ban vannak deklarálva ezért a program elején be kell “inkludolni” a “windows.h” fájlt.

Szoftvertechnikák

Felügyelt vastagkliens alkalmazások
fejlesztése



Automatizálási és
Alkalmazott
Informatikai Tanszék

Windows Forms alkalmazások architektúrája

Vastagkliens alkalmazások API vs .NET

- Win32 API ablakkezelés
 - > Natív, csak OS kell hozzá
 - > Alapvetően C és C++ (elvileg minden nyelvből lehet, ha támogatja)
 - > Nehéz jól strukturálni (switch-case szerkezet az üzenetek kezelésére)
 - > Nem objektumorientált
 - > Összetettebb vezérlőkkal (tree, list, tab) nagyon nehéz dolgozni
- Windows Forms
 - > .NET
 - > Kevesebb kód → hatékonyabb fejlesztés
 - > Több szolgáltatás

Architektúra (rétegek)

.NET Windows Forms alkalmazás

Windows
Forms

További .NET komponensek

.NET

Base Class Library (BCL)

Common Language Runtime (CLR)

Ablakok és
vezérlőelemek

Memória-
kezelés

Rajzolás

...

Win32
API

Az OS belső komponensei

Driverek

Részleges típusok (kitérő)

- Csak .NET 2.0-tól
- A fordító fésüli össze (nem lehetnek a részek külön szerelvényben)
- Fő területe: a generált és a kézzel írt kód különválasztása

```
public partial class Customer
{
    private int id;
    private string name;
    private List<Orders> orders
}
```

```
public partial class Customer
{
    public void SubmitOrder(Order order)
    {
        orders.Add(order);
    }

    public bool HasOutstandingOrders()
    {
        return orders.Count > 0;
    }
}
```

```
public class Customer
{
    private int id;
    private string name;
    private List<Orders> orders;

    public void SubmitOrder(Order order) {
        orders.Add(order);
    }

    public bool HasOutstandingOrders() {
        return orders.Count > 0;
    }
}
```

Alkalmazás architektúra

Demo

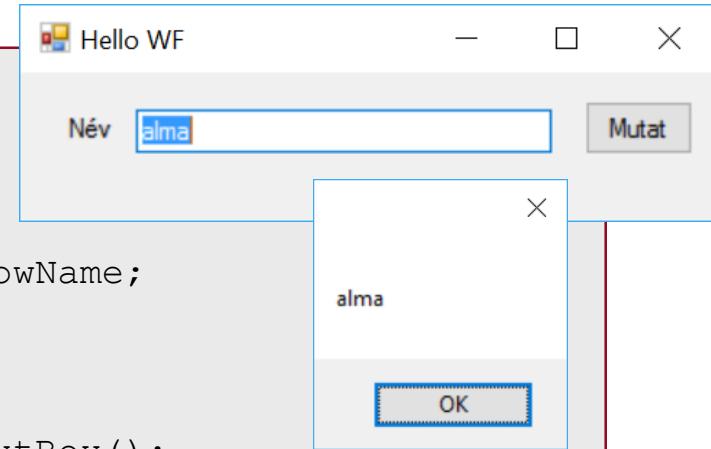
- System.Windows.Forms névtér
- minden ablak egy Form leszármazott osztály
- A Form
 - > Számos tulajdonsággal rendelkezik (pl. BackColor, Text, Size, ...)
 - > Számos eseményt publikál (Load – betöltéskor, Click – egérkattintás, Resize – átméretezés, Move – Mozgatás, KeyDown – billentyűlenyomás, ...)
- A Formon vezérlőelemeket helyezünk el (mint pl. TextBox, Label, stb.)
 - > Vagy vizuálisan a Visual Studio designer-ben, a ToolBox-ról
 - > Vagy programozottan
- A vezérlőelemek
 - > A Form leszármazott osztály tagváltozói lesznek
 - > Az konstruktorból hívott InitializeComponent-ben példányosítódnak
 - > Számos tulajdonsággal rendelkeznek (pl. Font property) és számos eseményt publikálnak (pl. a TextBox TextChanged-et)
- A Visual Studio részleges osztályokat generál

Demo

Demo

- A „Mutat” gombra kattintva beírt név megjelenítése

```
partial class MainForm
{
    private System.Windows.Forms.TextBox tbName;
    private System.Windows.Forms.Label Label1;
    private System.Windows.Forms.Button buttonShowName;
    ...
    private void InitializeComponent()
    {
        this.tbName = new System.Windows.Forms.TextBox();
        this.tbName.Location = new System.Drawing.Point(12, 25);
        this.tbName.Size = new System.Drawing.Size(100, 20);
        ...
        this.buttonShowName = new System.Windows.Forms.Button();
        this.buttonShowName.Location = new System.Drawing.Point(136, 22);
        this.buttonShowName.Size = new System.Drawing.Size(61, 23);
        this.buttonShowName.Click +=
            new System.EventHandler(this.buttonShowName_Click);
    }
}
```



Demo folytatás

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
    }

    private void buttonShowName_Click(object sender, EventArgs e)
    {
        MessageBox.Show(tbName.Text);
    }
}
```

Application.Run – Üzenetkezelő ciklust futtatja

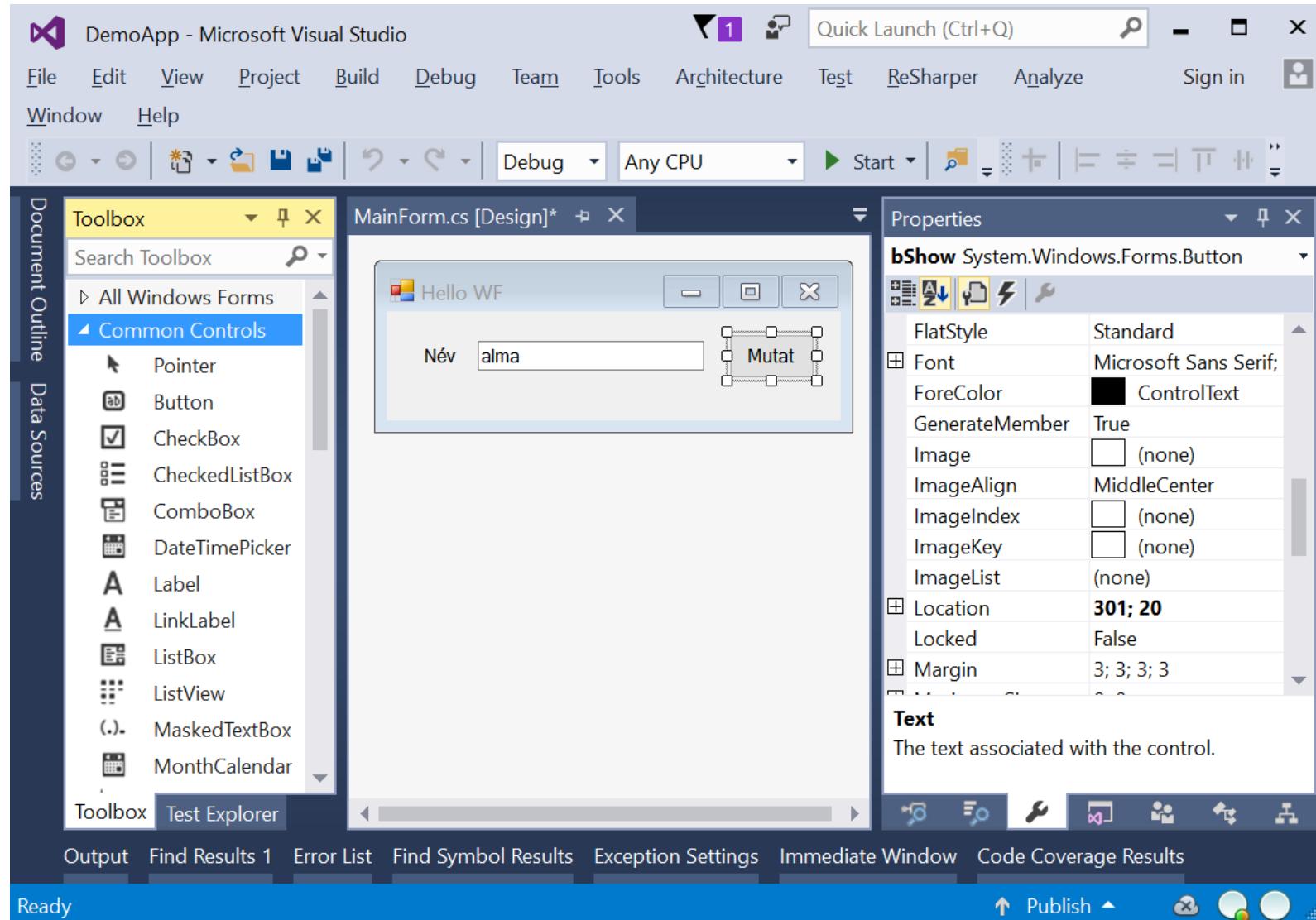
```
static class Program
{
    static void Main()
    {
        ...
        Application.Run(new MainForm());
    }
}
```

Eseményvezérelt programozás

Eseményvezérelt programozás:

- Megtervezzük a felhasználói felületeket
- A program működését az határozza meg, hogy hogyan reagálunk az egyes felhasználói eseményekre (pl. gomblenyomás, menükválasztás, egérkattintás, billentyűlenyomás), illetve az esetleges rendszereseményekre (pl. timer, lásd később).
- A Form és a rajta levő vezérlőelemek eseményeit a Form leszármazott osztályunkban kezeljük

A Visual Studio designere



Üzenet és eseménykezelés

- A .NET alkalmazások ráépülnek a natív Üzenet és ablakkezelésre
 - > Az ablakoknak és a vezérlőknek van ablakleírója (HWND, Control.Handle tulajdonság)
 - > A form egy csomagoló egy natív ablak körül
 - > A form és a vezérlők a Windows üzeneteket .NET eseményekké konvertálják
 - > Az alkalmazásnak van üzenetkezelő ciklusa (Application.Run futtatja)

Üzenet és eseménykezelés

Demo

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
        this.KeyDown += new KeyEventHandler(this.MainForm_KeyDown);
    }

    protected override void OnKeyDown(KeyEventArgs e)
    {
        base.OnKeyDown(e);
        MessageBox.Show("A billentyű (virt. fv.): "
            + e.KeyCode.ToString());
    }

    private void MainForm_KeyDown(object sender,
        KeyEventArgs e)
    {
        MessageBox.Show("A billentyű (eseménykez.): "
            + e.KeyCode.ToString());
    }
    ...
}
```

R

Az „R” billentyű lenyomása



Alkalmazás üzenetsorába
WM_KEYDOWN üzenet



A rejtett üzenetkezelő ciklus
kiveszi a sorból



Az ablak/form megkapja az
Üzenetet (WndProc)



virtual void OnKeyDown (
[KeyEventArgs](#) e) hívása



event [KeyEventHandler](#)
KeyDown elsütése

Fontosabb események

- A delegate típusa általában (ha nincs esemény paraméter)

```
public delegate void EventHandler (Object sender, EventArgs e)
```

- Object sender: az esemény kiváltója
- EventArgs e: esemény paraméterek
 - Az EventArgs nem hordoz információt
 - Ebből kell leszármaztatni, ha van esemény paraméter, pl. KeyEventArgs billentyű események esetében

Billentyű események

Esemény	Leírás	Delegate	Esemény paraméter
KeyDown	Billentyű lenyomás	void KeyEventHandler (Object sender, EventArgs e)	EventArgs.Shift – a Shift le van-e nyomva (ugyanígy Alt és Control) EventArgs.KeyCode – a billentyű módosító nélkül (Keys enum típus) EventArgs.KeyData - a billentyű a módosítóval (Keys enum típus)
KeyUp	Billentyű felengedés	Mint a KeyDown	Mint a KeyDown
KeyPress	Lenyomott billentyű esetén ismétlődő	void KeyPressEventHandler (Object sender, KeyPressEventArgs e)	KeyPressEventArgs.KeyChar – a lenyomott billentyű ASCII kódja, char típus

```
this_KeyDown += new KeyEventHandler(this.MainForm_KeyDown);  
...  
private void mainForm_KeyDown(object sender, EventArgs e)  
{  
    // Ha az ALT + E került lenyomásra  
    if(e.Alt && e.KeyCode == Keys.E) { ... }
```

Layout kezelés

- Lásd gyakorlaton ...
 - > Horgonyzás – anchor
 - > Dokkolás (ragasztás) – dock
 - > Splitter - SplitContainer

Menü, eszköz és statuszsáv

Demo

- Menüsáv

- Menüsáv
 - > ToolStrip
 - > Elemek: ToolStripMenuItem, ToolStripTextBox, ToolStripComboBox, separator

- Eszközsáv

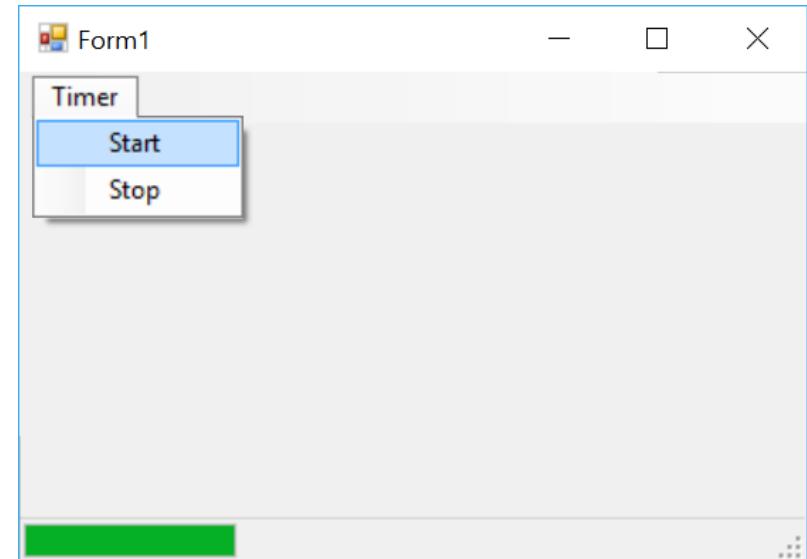
- Eszközsáv
 - > ToolStrip
 - > Elemek: ToolStripButton, stb.

- Státuszsáv

- Státuszsáv
 - > StatusStrip
 - > Elemek: ToolStripStatusLabel, stb.

- Lebegő menü

- Lebegő menü
 - > ContextMenuStrip



Időzítők

- **System.Windows.Forms.Timer** komponens
 - > Periodikus időzített esemény
 - > Interval tulajdonság: millisec-ben az időintervallum
 - > bool Enabled tulajdonság: timer tiltás/engedélyezés
 - > Start() és Stop() műveletek
 - > Tick event (EventHandler típusú) ha lejár az időzítő
 - > Pontatlan (a garantált minimum felbontás kb. 20 ms)
- **System.Threading.Timer** és **System.Timers.Timer**
 - > Pontos
 - > Többszálú környezetekben ajánlott

Forms.Timer használata példa

```
Timer timer = new Timer();  
  
void init()  
{  
    timer.Interval = 100;  
    timer.Tick += handleTimer;  
    timer.Start();  
}  
  
void handleTimer(object sender, EventArgs e)  
{  
    // Időzített feladat végrehajtása  
    // ...  
  
    // Ha nem szeretnénk ismétlődést, akkor állítsuk le:  
    // timer.Stop()  
}
```

Párbeszéd ablakok (dialog windows)

Demo

- Párbeszédblakok (=dialógusablakok) célja
 - > Tipikusan beállítások megjelenítésére, megváltoztatására
 - > Modális megjelenítés (nem lehet más ablakra átváltani)
- Új form felvétele: jobb katt. a projekten + Add/New Form
- Form keret stílus
 - > FormBorderStyle tulajdonság
 - FormBorderStyle.Sizeable (def.) - átméretezhető
 - FormBorderStyle.FixedDialog – nem átméretezhető
 - ...
- Párbeszédblak
 - > Az adatoknak tulajdonságok felvétele a Form osztályunkban
 - A példánkban Interval néven
 - > Az Form a DialogResult tulajdonságában jelzi, hogy érvényes-e az új érték (OK gombbal zárta-e be a felhasználó az ablakot). Lehetséges értékek:
 - DialogResult.Ok, DialogResult.Cancel, DialogResult.Yes, ...

Párbeszédablakok felépítése

```
public partial class SettingsForm : Form
{
    private int interval;
    public int Interval
    {
        get { return interval; }
        set
        {
            interval = value;
            textBox1.Text = value.ToString();
        }
    }

    private void bOk_Click(object sender, EventArgs e)
    {
        if (!int.TryParse(textBox1.Text, out interval))
            MessageBox.Show("Érvénytelen érték!");
        else
            this.DialogResult = DialogResult.OK;
    }
}
```

Párbeszédablakok megjelenítése

```
private void settingsToolStripMenuItem_Click(object sender,
    EventArgs e)
{
    SettingsForm form = new SettingsForm();
    form.Interval = timer1.Interval;
    if (form.ShowDialog() == DialogResult.OK)
    {
        MessageBox.Show("Changed!");
        timer1.Interval = form.Interval;
    }
}
```

- Beállítjuk a tulajdonságokat
- Modális megjelenítés: **Form.ShowDialog**
 - > Addig nem tér vissza, amíg be nem záródik a párbeszédablak.
- Ennek visszatérési értékét vizsgáljuk meg: Ok-kal zárták-e be az ablakot (**DialogResult** enum típus)

Üzenetablak - MessageBox

```
string message = "You did not enter a server name. Cancel this  
operation?";  
string caption = "No Server Name Specified";  
MessageBoxButtons buttons = MessageBoxButtons.YesNo;  
DialogResult result;  
  
result = MessageBox.Show(this, message, caption,  
    buttons, MessageBoxIcon.Question);  
if(result == DialogResult.Yes)  
{  
    this.Close();  
}
```



Nemmodális megjelenítés

- Aktiválható más ablak is
- Megjelenítés

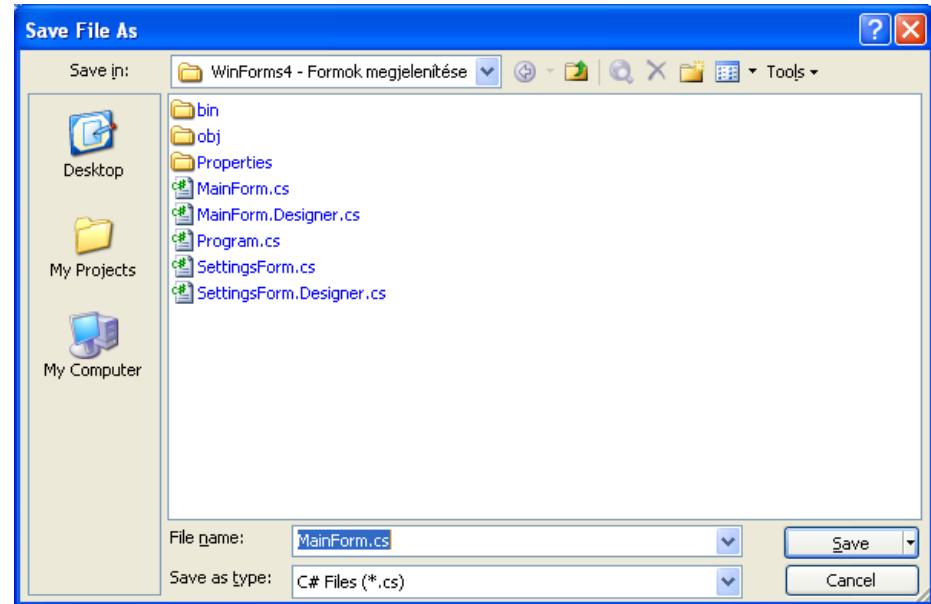
```
SettingsForm form = new SettingsForm();  
form.Show();
```

- A Show-nak megadhatunk egy owner (birtokos) ablakot
 - Nem kerülhet az owner (birtokos) ablak elő Z-orderben

```
SettingsForm form = new SettingsForm();  
form.Show(this); // A this egy Form leszármazott
```

Közös párbeszédablakok

- .NET csomagolók a Win32 API közös párbeszédablakok körül



```
OpenFileDialog ofd = new OpenFileDialog();
ofd.InitialDirectory = @"c:\";
ofd.Multiselect = false;
ofd.Filter = "C# Files (*.cs) | *.cs|All files (*.*) | *.*";

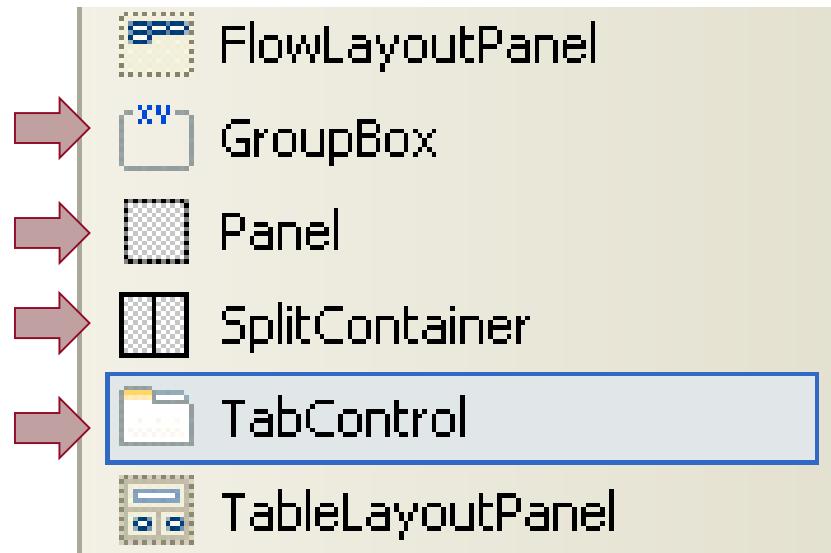
if ( ofd.ShowDialog() == DialogResult.OK )
{
    MessageBox.Show("A fájl útvonala: " + ofd.FileName );
}
```

Fontosabb *elemi* vezérlők

-  Button
 -  CheckBox
 -  CheckedListBox
 -  ComboBox
 -  DateTimePicker
 -  Label
 -  LinkLabel
 -  ListBox
 -  ListView
 -  MaskedTextBox
-  DataGridView

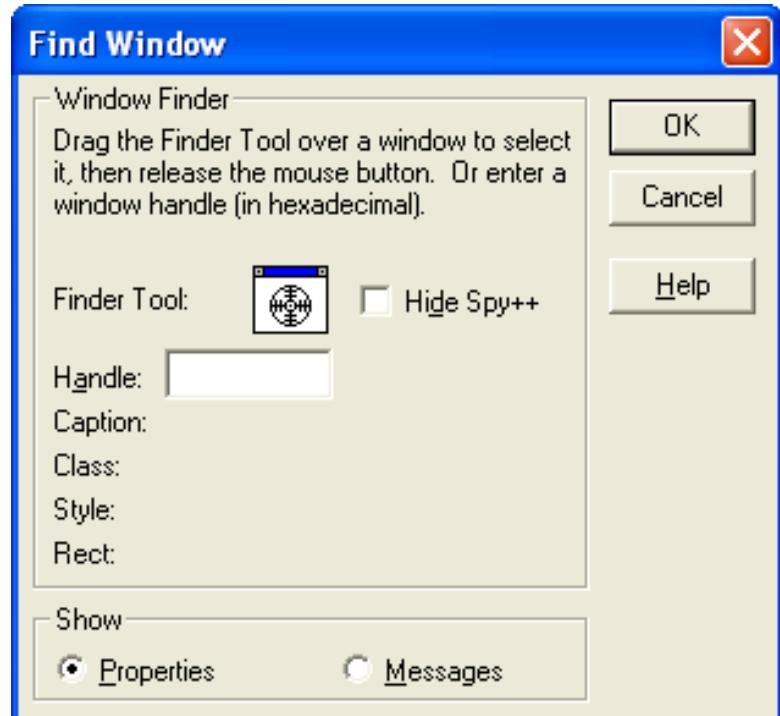
-  MonthCalendar
-  NotifyIcon
-  NumericUpDown
-  PictureBox
-  ProgressBar
-  RadioButton
-  RichTextBox
-  TextBox
-  ToolTip
-  TreeView
-  WebBrowser

Tároló vezérlők

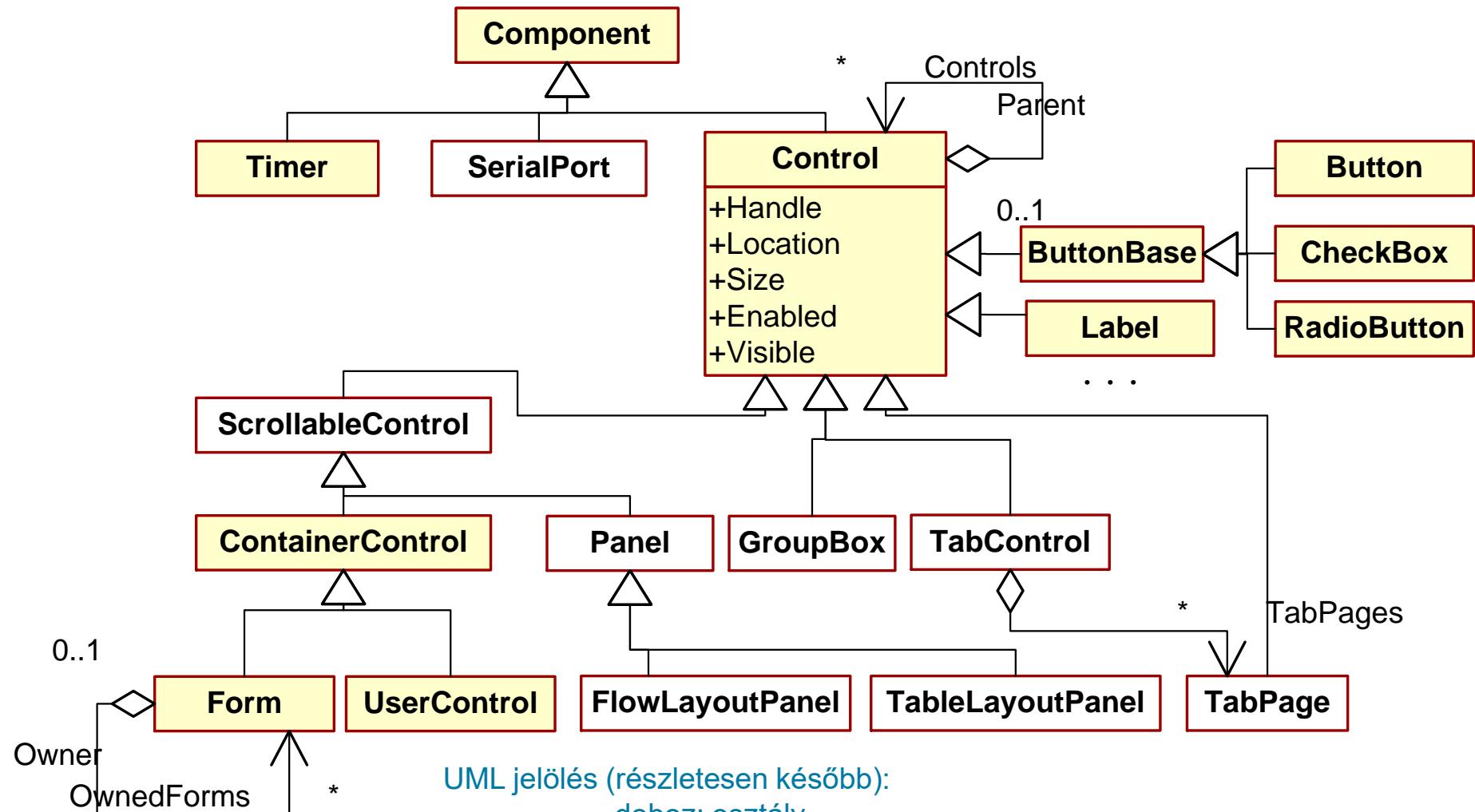


- Elemi, vagy további összetett vezérlők logikai csoportosítására
- A TabControlhozTabPage vezérlők adhatók

GroupBox példa:



Komponens/vezérlő hierarchia



UML jelölés (részletesen később):

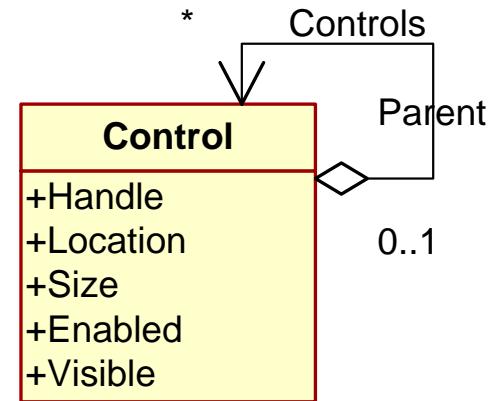
- doboz: osztály
- zárt háromszög végű nyíl: öröklés (ősre mutat)
- rombuszt tartalmazó nyíl: tartalmazás (tartalmazottra mutat)

Vezérlőhierarchia

- Component (komponens):
 - > Bármilyen, *container* (pl. *designer*) által tartalmazható komponens
 - > Nem feltétlenül vizuális (pl. SerialPort, Timer), de fel lehet dobni a designerbe, megadhatók vizuálisan a tulajdonságok és események
- Control (vezérlő):
 - > minden vezérlő ōse
 - > Natív ablak HWND tartozik hozzá (Handle tulajdonság)
 - > Összes közös tulajdonság
 - Visible
 - Enabled
 - Controls – Tartalmazott vezérlők lista (csak az összetett vezérlők)
 - Location, Size, ...
 - > Összes közös művelet
 - Focus(), ...
 - > Összes közös esemény
 - Click
 - KeyDown, ...

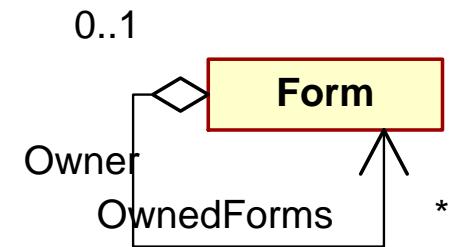
Vezérlők és Űrlapok - része hierarchia

- Szülő – gyerek viszony
 - > Control.Controls tulajdonság
 - A tartalmazott vezérlők gyerekablakok (child window)
 - > A megszokott viselkedést ez biztosítja
 - A gyerekablak együtt mozog a szülővel
 - Ha megszűnik a szülő, a gyerekablakok is automatikusan megszűnnek
 - A gyerekablak nem lóghat ki a szülőablakból (clipping)
 - A szülő elrejtése/megjelenítése automatikusan magával vonja a gyerekablakok elrejtését/megjelenítését
 - A szülőablak a Parent tulajdonsággal érhető el (null, ha nincs szülő)



Vezérlők és Űrlapok - része hierarchia

- Formok között más: birtokos-birtokolt (owner – owned) viszony
 - > Lazább, mint a parent-child
 - Ha bezáródik a birtokos, a birtokolt ablak is bezáródik (+ugyanez a minimize-ra)
 - A Z-orderben a birtokolt ablak a birtokos előtt helyezkedik el (?)
 - Birtokolt ablakok: *Form.OwnedForms* tulajdonság
 - Birtokos ablak: *Form.Owner* tulajdonság (null, ha nincs birtokos)



Amit még tudni kell

- TextBox
 - > Text tulajdonság
 - > TextChanged esemény (EventHandler)
- Timer
 - > Start, Stop műveletek.
 - > Interval tulajdonság
 - > Tick esemény (EventHandler)
- Label
 - > Text tulajdonság
- Button
 - > Click esemény
- CheckBox
 - > Checked tulajdonság
 - > Click esemény
- Form
 - > Text tulajdonság – fejléc
 - > KeyDown, KeyUp, KeyPress események
- Control
 - > Visible, Enabled tulajdonság

Ellenőrző kérdések

- Ismertesse ábrával illusztrálva a natív Win32 platform üzenetkezelését (ismétlés)! Ennek során térjen ki a következő fogalmakra: üzenet, üzenetsor, üzenetkezelő ciklus, ablakkezelő függvény, callback függvény, alapértelmezett ablakkezelő függvény!
- Ismertesse a .NET vastagkliens alkalmazások architektúráját (rétegek)!
- Ismertesse a .NET részleges típus (partial class) fogalmát és főbb felhasználási területét!
- Ismertesse a Windows Forms alkalmazások architektúráját!
- Ismertesse az üzenet és eseménykezelés kapcsolatát (pl. billentyűlenyomás esetére)
- Ismertesse a fontosabb események kezelésének menetét (EventHandler, EventArgs, billentyű események)!
- Hogyan lehet menüt és eszközsávot Windows Forms alkalmazásokban készíteni?
- Ismertesse a párbeszédablak fogalmát! Kódrészlettel illusztrálva ismertesse a párbeszédablakok kialakításának és megjelenítésének módszerét

Ellenőrző kérdések

- Ismertesse a nemmodális ablak fogalmát és kódrészlettel illusztrálja megjelenítését!
- Ismertesse a közös párbeszédablak fogalmát és ismertesse fajtáit!
- Sorolja fel a fontosabb elemi vezérlőket!
- Sorolja fel a fontosabb tároló vezérlőket!
- Rajzolja fel a komponens/vezérlő hierarchia fontosabb osztályait és kapcsolatukat! Ismertesse a fontosabb osztályokat (Component, Control, Form, ...)!
- Ismertesse a vezérlők és űrlapok közötti lehetséges viszonyokat (része hierarchia)!
- Ismertesse a fontosabb események kezelésének menetét (EventHandler, EventArgs, billentyű események)!

Ellenőrző kérdések

- Írjon olyan Windows Forms C# kódot, ami egy üzenetablakban megjeleníti a leütött billentyűt!
- Egy űrlapon egy timer1 nevű Timer időzítő komponens, egy label1 nevű címke, valamint egy mStart és mStop menuStrip elemekkel rendelkező MenuStrip van elhelyezve. Írjon olyan C# kódot, ami az mStart menüelem kiválasztásakor elindítja, az mStop menüelem kiválasztásakor leállítja az időzítőt. Ha az időzítő fut, másodpercenként eggyel növekvő egész számra állítsa a label1 címke szövegét. (Az eseménykezelők bekötéséről is gondoskodjon!)

Szoftvertechnikák

Windows Forms – saját vezérlők

Windows Forms – GDI+

Destruktor, Dispose



Automatizálási és
Alkalmazott
Informatikai Tanszék

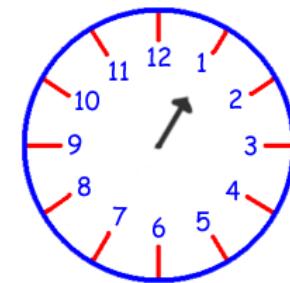
Saját vezérlők készítése

Saját vezérlők készítése

- Alapvetően nagyon egyszerű
- A cél az **újrafelhasználhatóság**
 - > A toolboxból akárhány helyre fel tudjuk dobni
- Hárromféle módszer
 - > Contol osztályból leszármaztatás
 - > Specifikus vezérlőből (pl. TextBox) leszármaztatás
 - > UserControl készítés
- Nézzük ezeket sorban ...

Leszármaztatás a Control alaposztályból

- Akkor használjuk, ha a egy teljesen új vezérlőelemet szeretnénk létrehozni
- Csak a minden Controlra közös tulajdonságokat kapjuk meg
- Adhatunk hozzá új tulajdonságokat (property), eseményeket (event)
- A rajzolás is a mi feladatunk (GDI+ -szal)
- Példa: egy az aktuális időt mutató vezérlő



Specifikus vezérlőből leszármaztatás

- Pl. TextBoxból, stb...
- Akkor használjuk, ha egy már létező vezérlőelemet szeretnénk testreszabni
- Csak a speciális viselkedést kell megvalósítani
- Adhatunk hozzá új tulajdonságokat, eseményeket
- Példa: egy speciális szövegablak, ami élénk háttérrel jelenik meg, ha érvénytelen e-mailcímét írt bele a felhasználó Demo

Származtatás a UserControl osztályból

- A vezérlőelem maga is egy űrlap, tartalmazhat vezérlőelemeket: tipikusan együtt előforduló vezérlőelemek összekötése
- Tervezési időben vizuálisan elkészíthetjük összetett vezérlőelemeinket, pont úgy, ahogy egy formot is „összekattintanánk”.
 - > Miben más? Toolboxon megjelenik: űrlapokra, illetve más UserControlokra lehet elhelyezni.

Demo

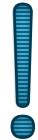
- Példa saját „FilePicker” vezérlő:



- A tartalmazott vezérlőelemek private láthatóságúak – így logikus
- Hogyan induljuk el? Visual Studio: Add New Item/User Control

A UserControlok szerepe

- Mikor készítsünk saját UserControl-t? Két cél:
 - > Újrafelhasználás
 - > Összetett felhasználói felület modularizálásának eszköze!



The screenshot shows the IssueVision application interface. The main window has a title bar "IssueVision - demo" and a menu bar with "File" and "Help". Below the menu is a toolbar with icons for "New", "Send/Receive", and "Work Offline".

The left side features a "Views" panel with sections for "Staff list" (containing a tree view of staff members labeled 'b') and "Issues resolved" (containing a chart with categories like "Resolved" (10), "Escalated" (2), and "Open" (3) labeled 'a').

The central area is titled "Issues" and contains a list of open issues under "Telecommunications" and "Computer" categories. The "Computer" category is expanded, showing three specific issues labeled 'c'. The right side is a detailed view of the first "Computer" issue, titled "Computer won't shut down". It includes sections for "Issue Details" (Status: Open, Age: 12/29/2003, Category: Computer, Assigned To: Adam Barr) and "History" (User: Adam Barr, Comment: Issue created, Date: 12/29/2003 4:03 pm). A blue bar at the bottom right says "Working Online".

Destruktor, Finalize és Dispose

A destruktör szerepe C++-ban

- Nincs szemégtgyűjtő (garbage collector)
- Általában az osztály feladata az általa dinamikusan lefoglalt memória felszabadítása

```
class Stack {  
private:  
    int* pFirst; int* pCurrent; int size;  
public:  
    void push(int n) { ... } ;  
    int pop() { ... } ;  
    Stack(int size) : size(size)  
    {  
        pCurrent = pFirst = new int[size];  
    }  
    ~Stack() { delete[] pFirst; }  
};
```

.NET - Finalize és a destruktur

- Nincs delete, a már nem hivatkozott objektumokat a szemétgyűjtő (garbage collector - GC) szabadítja fel

```
public class Stack
{
    int[] items;
    public Stack(int size)
    {
        items = new int[size];
    }
    ...
    ~Stack()
    {
        delete items; // NEM LÉTEZIK, nincs is szükség rá!
    }
}
```

- Ha egy Stack objektumra nem hivatkozik más objektum, akkor az int[] items-re sem: a GC ezt is begyűjti.
 - > Nincs is szükség a destruktorra... - de soha nincs?

.NET - Finalize és a destruktur

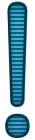
- A GC általi felszabadítás során meghívódik az objektumra a Finalize művelet (egységes CLR név)
 - > ez az ún. Finalizer
 - > Aminek C# nyelven a destruktur felel meg (~osztálynév {...})! C# nyelven nem definiálhatjuk felül a Finalize nevű műveletet! Az alábbi kód

```
class MyClass{  
    ~MyClass() { // Takarítás ... }  
}
```

- > ezzel ekvivalens:

```
class MyClass {  
    protected override void Finalize()  
    {  
        try  
        { // Takarítás ... }  
        finally { base.Finalize(); }  
    }  
}
```

Finalizer és destruktur jellemzők



- Finalizer/destruktur jellemzők
 - > A végrehajtás ideje nem ismert! (amikor a CLR úgy gondolja, ideje futtatni a GC-t)
 - > A sorrend nem ismert
 - Ha pl. 100 objektum szabadítható fel, milyen sorrendben szabadulnak fel (ezek hivatkozhatnak is egymásra!)...
 - > A szál nem ismert!
- Megjegyzés
 - > A destruktoral rendelkező osztályok megszűnés előtt bekerülnek a finalizer queue-ba: lassít!

Destruktor (Finalize) jellemzők

- Ez rendben van?

```
public class Stack
{
    int[] items;
    public Stack(int size)
    {
        items = new int[size];
    }
    ...
    ~Stack()
    {
        Console.WriteLine(items.Length);
    }
}
```

- Lehet, hogy az `items` objektumot előbb gyűjti be a GC, mint a `Stack`-et: ekkor a destruktur egy olyan objektumot használ, aminek már meghívódott a destruktora! Ne tegyük!
- Akkor soha ne is írunk destruktort??? →

A destruktur létjogosultsága

- Példa

```
public class MySuperFileReader
{
    // Natív fájlleíró - egy nem felügyelt erőforrás
    private IntPtr fileHandle;

    public MySuperFileReader(String fileName) {
        // Egy natív DLL függvényt hívunk, ami
        // megnyitja a fájlt.
        fileHandle = FileFunctions.CreateFile(fileName, ...);
        ...
    }

    ~MySuperFileReader() {
        // Egy natív DLL függvényt hívunk, ami
        // lezárja a fájlt.
        FileFunctions.CloseHandle(fileHandle);
    }
}
```

A destrukturor létjogosultsága

- A „MySuperFileReader” ún. NEM FELÜGYELT erőforrást használ (esetünkben egy fájlleírót). Ezeket a GC NEM gyűji be, felszabadításuk a mi feladatunk. Az ezért felelős kódot tettük a destrukturorba.
- Mit értünk el?
 - > A nem felügyelt erőforrások nem szivárognak el, nem maradnak felszabadítatlanul. Ez által nyer létjogosultságot a destrukturor .NET környezetben.
- Nem felügyelt erőforrás pl.:
 - > Fájl
 - > Nem felügyelt lockok, mutexek
 - > Adatbázis-kapcsolat
 - > minden natív ablak
 - > Vagyis minden, amit a .NET alatti OS-ben közvetlenül foglalunk le.

Finalizer és destruktur összefoglalás

- Mikor írunk destruktort?
 - > Láttuk - memóriát nem kell persze felszabadítani...
 - > Csak ha nem felügyelt erőforrást foglal az objektum (pl. natív ablak leíró, adatbáziskapcsolat, file, lock, stb.) .
 - > Egyébként ne írunk, mert a destruktoral rendelkező osztályok megszűnés előtt bekerülnek a finalizer queue-ba: lassít!
- Finalize/destruktur jellemzők
 - > A végrehajtás ideje nem ismert!
 - > A sorrend nem ismert!
 - > A szál nem ismert!
 - > A destruktur csak a külső hivatkozásait engedheti el (pl. nem felügyelt erőforrások, mert a felügyelteket már lehet felszabadította a szemétgyűjtő)
 - A TAGVÁLTOZÓKHOZ, MELYEK FELÜGYELT ERŐFORRÁSRA HIVATKOZNAK (közönséges .NET objektumokra), ENNEK MEGFELŐEN NE IS FÉRJÜNK HOZZÁ a destruktorból!

A destruktur létjogosultsága

- Mit értünk el a destruktoral?
 - > A nem felügyelt erőforrások nem szivárognak el, nem maradnak felszabadítatlanul.
- Önmagában elég a destruktur?
 - > Mi a helyzet a „drága” erőforrásokkal: file, adatbáziskapcsolat, lock, stb.?
 - > A destruktur lefutása nemdeterminisztikus!
Példa: Ha egy fájlleírót csak a destrukturban szabadítunk fel, lehet órákat kell várni arra, hogy a GC meghívja a destruktort: a fájlunk addig zárolt lesz, önmagában nem jó megoldás.
- A „drága” nem felügyelt erőforrásokat explicit, azonnal fel szeretnénk szabadítani, ha már nem használjuk!
Mognézzük, hogyan ... →

Determinisztikus erőforrás felszab.

- A szemégtgyűjtés kikényszeríthető

```
System.GC.Collect();
```

- Csak a legritkább esetben (soha) hívjuk, bízzunk a CLR-ben!
 - > De ennél sem garantált, hogy minden erőforrást felszabadít!
- Helyette mi a gyakorlatban használatos megoldás a drága erőforrások determinisztikus felszabadítására?
 - > A **Dispose** minta alkalmazása! →

A Dispose minta lényege

- A nem felügyelt erőforrást foglaló osztály implementálja az **IDisposable** interfészt.
- Ennek egy művelet van, a **Dispose()**. Ebben kell felszabadítani a nem felügyelt erőforrást.
 - > Pl. ilyen a szövegfájlok olvasására szolgáló StreamReader és az írásra szolgáló StreamWriter osztály.
- A destuktoral kombinált megoldás: lásd IDisposable.Dispose leírása az MSDN Library-ben (nem kell tudni).
 - > A destrukturban is gondoskodjuk a nem felügyelt erőforrások felszabadításáról, hogy akkor se szivárogjanak el, ha a fejlesztő elfelejtett Dispose-t hívni.
- Sok esetben beszédedesebb néven is elérhető a Dispose() művelet, pl. File esetén a Close() művelet Dispose()-t hív.

Dispose használata

- Viszonylag ritkán készítünk IDisposable-t implementáló osztályt
- Annál gyakrabban használunk (pl. StreamReader, stb.)
- Az IDisposable-t implementáló osztályok használata
- Garantált Dispose hívás kell, kivétel esetén is! Így nem jó:

```
ResourceWrapper r1 = new ResourceWrapper();  
... // r1 használata  
R1.Dispose(); // Így NEM hívódik, ha az előző sorokban  
              // kivétel volt!
```

Dispose használata – try - finally

- A ResourceWrapper implementálja az IDisposable-t

```
ResourceWrapper r1 = new ResourceWrapper();  
try  
{  
    // r1 objektum használata  
    r1.DoSomething();  
}  
finally  
{  
    // null feltétel ellenőrzés  
    if (r1 != null) r1.Dispose();  
}
```

Dispose használata - using

- Alap

```
using (ResourceWrapper r1 = new ResourceWrapper())
{
    // r1 objektum használata
    r1.DoSomething();
} // Kilépünk a using blokkból: meghívódik r1-re a Dispose!
```

- Több egyszerre

```
using ( StreamReader s1 = new StreamReader(path1) )
using ( StreamReader s2 = new StreamReader(path2) )
{
    // Az s1 és s2 StreamReader objektumok használata
} // Kilépünk a using blokkból: meghívódik s1, s2-re a Dispose!
```

A Dispose és a vezérlők

- A Component osztály (a Control őse) implementálja az IDisposable interfészt
 - > Ebből következik: minden vezérlő implementálja az IDisposable interfészt
 - > Egy natív ablak (HWND-je van) drága erőforrás
 - > A Form Close() bezárja az űrlapot
 - Form.Show() megjelenítést követően Dispose()-t is hív
 - Form.ShowDialog() esetén nekünk célszerű ezt megtenni
- A Dispose mintának megfelelően a Form leszármazott osztályunk
 - > Lezárja a natív ablakot az ős Dispose hívásával
 - > minden tartalmazott vezérlőre (Controls tulajdonság) Dispose-t hív, ami lezárja a vezérlő mögötti natív ablakot
 - > minden tartalmazott komponensre (component tagváltozóban tárolt), ami nem vezérlő (pl. Timer) Dispose-t hív
- A Bitmap, Graphics, Pen és Brush is implementálja az IDisposable-t

A destruktur és a dispose kapcsolata

- Destruktor
 - > Garantálhatóvá teszi, hogy a nem felügyelt erőforrások felszabadításra kerüljenek (legkésőbb akkor, amikor a nem felügyelt erőforrás csomagoló osztályát a GC begyűjti)
- Dispose minta
 - > Egységes kezelést biztosít nem felügyelt erőforrások determinisztikus felszabadítására
- Kombináltan szokás használni

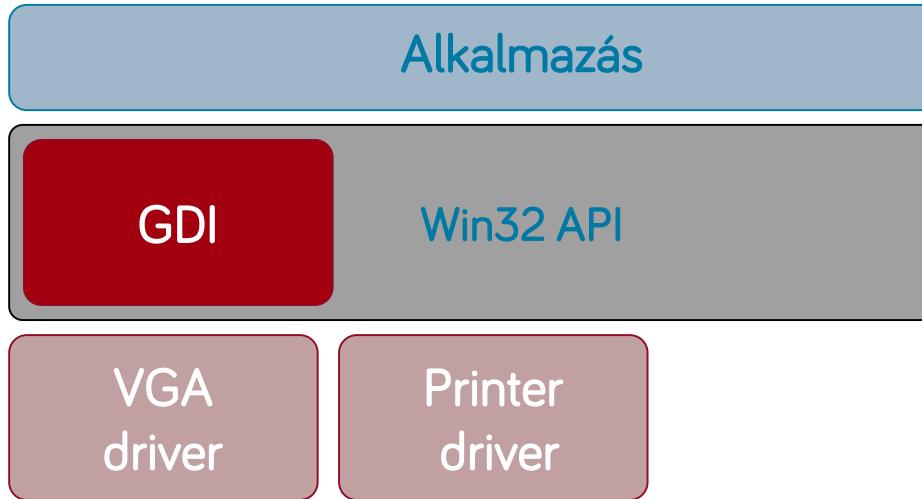
Grafikus megjelenítés natív környezetben

Grafikus megjelenítés natív környezetben

- GDI – Graphical Devices Interface
- Grafikus megjelenítés natív Win32 alkalmazások esetén
- Mit nyújt
 - > Vektorgrafikus műveletek: pont, téglalap, stb. megjelenítése
 - > Csak 2D!
 - > Szöveg megjelenítése
 - > Bitmap megjelenítése
 - > Metafile támogatás
 - > Koordináta transzformációk
 - > ...

GDI architektúra

- Eszközfüggetlen grafikus megjelenítést támogat
 - > Felbontás és színmélység függetlenül rajzolunk
 - > „Ugyanúgy” rajzolunk képernyőre, mint nyomtatóra
 - Nyomtatásnál mi a feladatunk -> oldalakra tördelés



- Az eszközfüggetlenség alapja a **Device Context**, vagyis **eszközkapocsolat**

Mi az eszközkapcsolat (Device Context)?

- Reprezentál egy grafikus eszközt
- HDC leíró azonosítja
- Lépések
 - > 1. Eszközkapcsolat létrehozás
 - > 2. Rajzolás az eszközkapcsolatra
 - > 3. Eszközkapcsolat lezárás
- A DC „rajzolófelület”: minden rajzoló függvénynek ez az első paramétere
- Eszközkapcsolat létrehozható
 - > Ablak kliens területre
 - > Teljes ablakra (pl. fejlécre, statuszbarra, menüre, stb., is tudunk rajzolni)
 - > Képernyőre ☺
 - > Memóriába rajzolásra
 - > Nyomtatóra
 - > Metafilera
 - > Érvénytelen területre

Rajzolás példa

- Ablak kliens területre rajzolás esete
 - > Nem kell tudni a kódot

```
HDC hdc;  
hdc = GetDC(hWnd); // DC létrehozás adott ablakhoz  
Rectangle( hdc, 10, 10, 20, 20 ); // Rajzolunk rá  
ReleaseDC(hdc); // DC felszabadítás
```

Érvénytelen terület

!

- Az OS a rajzot nem jegyzi meg!
- Ha érvénytelen terület keletkezik, újra kell rajzolni azt!
- Érvénytelen terület:
 - > Korábban takarásban levő, láthatóvá vált ablakrészeken (pl. átméretezés, Z-orderben előbb került az ablak, stb.)
- Miért nem jegyzi meg a rajzot az OS?
 - > Sok ablak létezhet egyszerre -> memóriakorlát!
- Honnan tudjuk, hogy érvénytelen terület keletkezett egy ablakon?
 - > WM_PAINT üzenetet kap az ablak

Érvénytelen területre rajzolás

- Így általában a WM_PAINT üzenet kezelésében rajzolunk

```
HRESULT CALLBACK WndProc(HWND hWnd, UNIT message,
    WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    switch(message)
    {
        case WM_PAINT:
            hdc = BeginPaint(hWnd, &ps); // DC létrehozás
            Rectangle(hdc, 10, 10, 20, 20); // Rajzolás
            EndPaint(hWnd, &ps); // DC felszabadítás
            break;
            ...
    }
}
```

- Vagy duplapuffereléssel: elkészítjük előre a rajzot egy memória eszközkapcsolaton és a WM_PAINT-ben csak másolunk róla

Grafikus megjelenítés felügyelet környezetben

Architektúra

- A .NET a GDI+ -t támogatja (GDI +: számos extra szolgáltatás)
 - > Nagyon hasonlít a natív Win32 API modellhez
 - > **System.Drawing** névtér és szerelvény
 - > Eszközkapcsolat (DC) helyett **System.Drawing.Graphics** objektumra rajzolunk, ami egy rajzolófelületet reprezentál
- Érvénytelen területre rajzolás, mint natív esetben
- Mi hogyan kezelhetjük (vagyis hol rajzolunk a kódban tipikusan)?
 - > Űrlap/vezérlőelem **Paint** eseményéhez eseménykezelőt rendelünk
 - > Vagy felülbíráljuk (override) az **OnPaint** műveletét

Legegyszerűbb példa

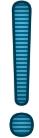
- Az Form leszármazott osztályunkban:

```
protected override void OnPaint(PaintEventArgs e)
{
    e.Graphics.FillRectangle(
        new SolidBrush(Color.Blue), 10, 10, 100, 100);
}
```

- **PaintEventArgs .Graphics objektum: erre tudunk rajzolni**
- PaintEventArgs .ClipRectangle: újrafestendő terület (ritkán használjuk)
- Form.ClientRectangle: ablak kliens területe (ritkán használjuk)
- Nemcsak az űrlapoknál, hanem saját vezérlőelemeknél is ugyanígy rajzolunk!

Az OnPaint-tet ne hívjuk explicit (az OS optimalizál)!

> Helyette: Invalidate-et hívunk, mely érvényteleníti a területet és kiváltja az újrarajzolást!



Színkezelés

- *Color* struktúra
 - > **Color.Red** – egyszerű szín, itt piros
 - > **Color.FromArgb**
 - 4 színösszetevő (alpha, red, green, blue), a tartomány mindenre 0-255
 - `Color.FromArgb(127, Color.Red);` // Félig átlátszó piros szín
 - `Color.FromArgb(100, 255, 0, 0);` // Félig átlátszó piros szín
 - > **Color.Empty** – üres szín (null-nak felel meg)
 - > **Color.Transparent** – A transzparens színt reprezentáló szín (tipikusan háttérszínnek szokás megadni) – lásd később

Rajzolóműveletek

- A Graphics osztály műveletei
 - > `Graphics.DrawRectangle` (Pen, Rectangle) - téglalap
 - > `Graphics.DrawRectangle` (Pen, Int32, Int32, Int32, Int32)
 - > `Graphics.DrawLine`(Pen, Point, Point)
 - > `Graphics.FillRectangle` (Brush, Rectangle) - kitöltött téglalap
- Eddig kell tudni, de nagyon sok egyéb:
 - > DrawBezier – Bezier görbe
 - > DrawCurve – Spline (pontokhoz illesztett görbe)
- ...

Rajzolóműveletek

- Rajzolóműveletek csoportosítása
 - > Draw kezdetűek: körvonalas rajz, tollal (pen)
 - > Fill kezdetűek: kitöltött, ecsettel (brush)

Toll (pen)

- A toll vonalak színét, mintáját, vastagságát határozza meg
- Pen osztály
 - > Példa 1 - kitöltetlen téglalap adott tollal

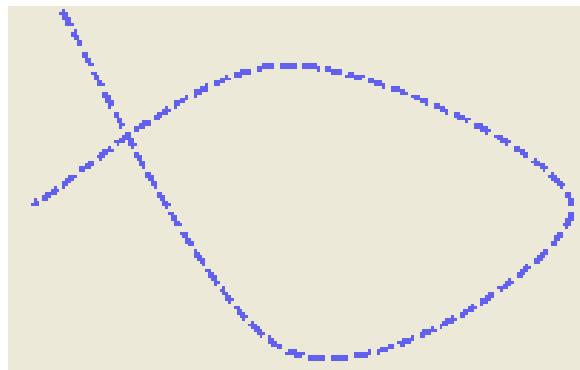
Graphics osztály művelete:

```
public void DrawRectangle ( Pen pen, int x, int y,  
    int width, int height )
```

- > Példa2

```
protected override void OnPaint(PaintEventArgs e) {  
    e.Graphics.DrawLine( new Pen(Color.Red), 10, 20, 220, 20);  
}
```

- > Példa3



Toll

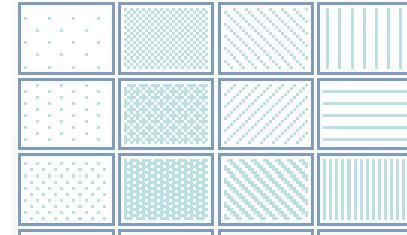
```
protected override void OnPaint(PaintEventArgs e)
{
    // Konstruktor paraméter: a vonal színe és vastagság
    Pen pen = new Pen(Color.FromArgb(150, Color.Blue), 2)
    // Vonal minta (ha nem adjuk meg, folytonos)
    pen.DashStyle = System.Drawing.Drawing2D.DashStyle.Dash;
    // Vonal lezárása (ha nem adjuk meg, nincs)
    pen.EndCap = LineCap.ArrowAnchor;

    // spline
    g.DrawCurve(pen, new Point[] {
        new Point(10, 110), new Point(100, 250),
        new Point(200, 200),
        new Point(100, 150), new Point(10, 200)} );
}
```

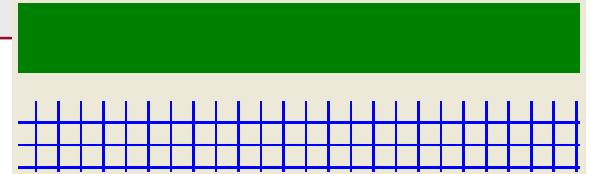
- Ilyen részletességgel nem kell tudni
- Amit tudni kell: Pen használata DrawLine és DrawRect esetén.
- Vannak előredefiniált tollak, pl. **Pens.Blue** – kék folytonos 1px vastag vonal
 - > Egyszerű: nem kell létrehozni, felszabadítani!

Ecset (brush)

- Az ecset az alakzatok kitöltési színét, mintáját határozza meg
- Brush œsosztály, leszármazottak
 - > SolidBrush – adott színnel „tele” kitöltés
 - > HatchBrush – vonalmintával kitöltés, pl.
 - > TextureBrush – bitmintával kitöltés
 - > LinearGradientBrush – lineáris színátmenet
 - > ...
- Példa 1



```
protected override void OnPaint(PaintEventArgs e) {  
    e.Graphics.FillRectangle(new SolidBrush(Color.Green),  
        10, 30, 200, 25);  
    e.Graphics.FillRectangle(new HatchBrush(HatchStyle.Cross,  
        Color.Blue, Color.Transparent), 10, 65, 200, 25);  
}
```



Ecset

- Példa2 - LinearGradientBrush

```
protected override void OnPaint(PaintEventArgs e)
{
    Rectangle r = new Rectangle(10, 100, 200, 25);
    g.FillRectangle( new LinearGradientBrush(
        r, // Az ecset befoglaló téglalapja
        Color.Red, // Kiinduló szín
        Color.Yellow, // Cél cím
        LinearGradientMode.BackwardDiagonal), // Stílus
        r);
}
```



- Csak annyit kell tudni, hogy ezt is támogatja
- Vannak előredefiniált ecsetek:
 - Pl. **Brushes.Blue** – kék tele ecset
 - Egyszerű használat: nem kell létrehozni, felszabadítani!

Szöveg megjelenítése

- Egyszerű, az űrlap betűtípusát használva, (10, 10) pontban

```
protected override void OnPaint(PaintEventArgs e)
{
    e.Graphics.DrawString("Hello World", this.Font,
        new SolidBrush(Color.Black), 10, 10);
}
```

- Téglalap bal felső sarkában (ami nem fér, levágja)

```
protected override void OnPaint(PaintEventArgs e)
{
    e.Graphics.DrawString("Hello World",
        new Font("Lucida Sans Unicode", 8),
        new SolidBrush(Color.Blue),
        new RectangleF(100, 100, 250, 350)); // Befoglaló tégl.
}
```

Szöveg megjelenítés

- A StringFormat-tal „mindent” lehet

```
string text = "Hello World";  
  
StringFormat format = new StringFormat();  
// Igazítás (Near - balra, Center, Far - jobbra)  
format.Alignment = StringAlignment.Center;  
// Horizontális igazítás (Near - fent, Center, Far - lent)  
format.LineAlignment = StringAlignment.Center;  
// Nincs sortörés.  
format.FormatFlags = StringFormatFlags.NoWrap;  
// Ha nem fér ki a szöveg: elvágja és ...-tal zárja le!  
format.Trimming = StringTrimming.EllipsisCharacter;  
  
g.DrawString( text, new Font("Arial", 16, FontStyle.Bold),  
    new SolidBrush(Color.Blue),  
    new RectangleF(100, 100, 250, 350),  
    format);
```

- Nem kell tudni a szintaktikát

Példa 1

Demo

- Feladat 1. Számoljuk az ablak közepén, hogy az ablak hányszor rajzolja újra magát (mi a kapott WM_PAINT-ek száma)!

- > Úrlap átméretezés: alapértelmezésben nem festi újra magát. Engedélyezése (a form/control stílusát kell állítani, pl. a konstruktorban):

```
this.SetStyle(ControlStyles.ResizeRedraw, true);  
UpdateStyles();
```

- > Ekkor meg villog átméretezéskor. Engedélyezzük a duplapufferelést az űrlapra (a konstruktorban)!

```
DoubleBuffered = true;
```

Példa 1

- Megoldás – lásd Visual Studio solution
- Alapelve
 - > Az **OnPaint** műveletben növelünk egy számlálót
 - > Az **OnPaint** műveletben ki is rajzoljuk az értékét a `DrawString` művelettel

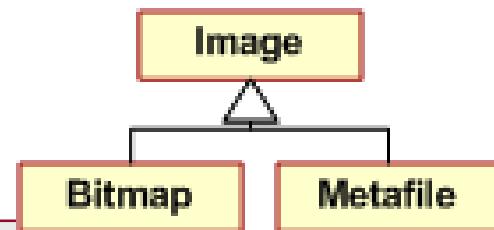
Példa 2

Demo

- Feladat2. Az ablak közepén másodpercenként eggyel növeljünk egy számlálót!
- Megoldás – lásd Visual Studio solution
- Alapelve
 - > Timer Tick eseményében növelünk egy számlálót
 - > Timer Tick eseményében Invalidate()-et hívunk
 - Kiváltja az OnPaint eseményt
 - > Az OnPaint műveletben rajzoljuk ki a számláló értékét a DrawString művelettel
- Megjegyzés: OnPaint-tet soha NEM hívunk: helyette Invalidate()!

Kép megjelenítése

- Bitmap - Bitmap osztály, **raszteres**



```
protected override void OnPaint(PaintEventArgs e)
{
    using (Bitmap bmp = new Bitmap("wheel.png")) // Fájl útvonal
    {
        // (0,0) pont definiálja a transzparens színt
        if (checkBoxTransparentImage.Checked)
            bmp.MakeTransparent(bmp.GetPixel(0, 0));
        e.Graphics.DrawImage(bmp, 250, 70, bmp.Width / 2, bmp.Height / 2);
    }
}
```

Nem transzparens
megjelenítés:

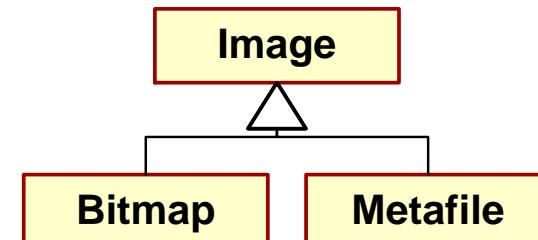


Transzparens
megjelenítés:



- Metafile – Metafile osztály

Előre rögzített, lejátszható **vektorgrafikus**
műveletek. A kimenete egy kép, de
nagyításkor nem pixelesedik.



Manuális duplapufferelés

- Ha maga a rajzolás lassú (pl. bonyolult rajz, ábra), a megjelenítés villog, átméretezéskor akadozik. Oka: minden OnPaint esetén lefut a lassú megjelenítő algoritmus.
 - > Vegyük észre: ezen a beépített duplapufferelés engedélyezése sem segít.
 - > Megoldás: manuális duplapufferelés (virtuális ablakok módszere). Alapelve: egy memória **Graphics** objektumra rajzolunk, az **OnPaint-ben** ennek tartalmát másoljuk a képernyőre (ez nagyon gyors művelet).
Így ha sok OnPaint esemény keletkezik is, gyorsan le fog futni mind: nem villog, nem szaggat.

Koordináta transzformációk

- A rajzolásra tetszőleges lineáris transzformáció alkalmazható

```
protected override void OnPaint(PaintEventArgs e)
{
    Graphics g = e.Graphics;
    // Transzformációs mátrix
    Matrix m = new Matrix();
    // Beállítjuk a (80, 270) koordinátapont körüli elforgatásnak
    // megfelelően.
    m.RotateAt(30, new PointF(80, 270));
    // Aktiváljuk a Graphics objektumra a transzformációs mátrixot.
    g.Transform = m;

    g.DrawString("HELLO!",
        new Font("Arial Black", 40, FontStyle.Bold),
        new TextureBrush(
            new Bitmap("Santa Fe Stucco.bmp")),
        new PointF(10, 260)
    );

    g.ResetTransform();
}
```



Mit tud még a GDI+

- **Anti-aliasing**
- Beépített támogatás **.jpeg**, **.png**, **.gif**, .bmp (kiterjeszthető) formátumokhoz
- Képfeldolgozás: világosság, kontraszt, stb.
- Vonal stílusok 
- Teljes Unicode támogatás (szöveg tetszőleges nyelven)
- Lebegőpontos számokkal is dolgozhatunk:
- Illetve RectangleF és PointF típusok
- Tetszőleges alakú űrlapok készíthetők.
- **Nyomtatás!**
- Sok minden más...
- 3D-t nem (ahhoz pl. DirectX kell)

Windows Forms összefoglalás

- A Windows Forms vastag kliens alkalmazások hatékony fejlesztését teszi lehetővé
- Nem tökéletes:
 - > Nehéz vele DPI független alkalmazásokat fejleszteni
- Újabb technológiák
 - > Windows Presentation Foundation, WPF (a .NET 3.0 -tól)
 - > Universal Windows Application, UWP (Windows 10-től)
- Amiről nem volt szó a Windows Forms kapcsán
 - > Validálás
 - > Adatkötés
 - > Nyomtatás
 - > ...

Ellenőrző kérdések

- Grafikus megjelenítés
 - > Ismertesse a GDI architektúráját! Mit jelent az eszközfüggetlen grafikus megjelenítés?
 - > Ismertesse az eszközkapcsolat fogalmát!
 - > Ismertesse a megjelenítés mechanizmusát natív környezetben! (Érvénytelen terület, WM_PAINT üzenet, stb.)
 - > Kódrészettel is illusztrálva ismertesse a megjelenítés mechanizmusát felügyelt környezetben! (Érvénytelen terület, Paint esemény, OnPaint virt. fv., Graphics osztály, ...)
 - > Ismertesse a színkezelés alapjait!
 - > Ismertesse a Pen és a Brush fogalmát, röviden a típusait és egy egyszerű példával illusztrálja használatukat!
 - > Ismertesse a metafájl fogalmát!
 - > Ismertesse a szöveg megjelenítésének lehetőségeit! Mutasson példát szöveg adott koordinátában való megjelenítésére!
 - > Mi a manuális duplapufferelésen alapuló megjelenítés lényege?

Ellenőrző kérdések

- Grafikus megjelenítés
 - > Példa: Írjon olyan C# nyelvű alkalmazást, ami a (10,10) koordinátában megjeleníti az ablakfrissítések számát!
 - > Példa: Írjon olyan C# nyelvű alkalmazást, ami a (10,10) koordinátában másodpercenként eggyel növelve megjeleníti egy számláló értékét!
 - > Írjon olyan C# nyelvű alkalmazást, ami a (10,10) koordinátában másodpercenként eggyel növelve megjeleníti egy számláló értékét!
 - > Írjon olyan C# nyelvű alkalmazást, ami másodpercenként felváltva megjelenít egy tele kék és zöld színnel kitöltött négyzetet a (10,10) pontban 20 pixel oldalhosszúsággal!
 - > Írjon olyan C# nyelvű alkalmazást, amely a (10,10) pontban egy piros, 1 pixel vastag folytonos vonallal rajzolt 10 pixel oldalhosszúságú négyzetet jelenít meg. A négyezet a kurzorbillentyűkkel lehessen mozgatni! (A kurzorbillentyűk kódja: Keys.Up, Key.Left, ...)

Ellenőrző kérdések

- Saját vezérlők készítése
 - > Ismertesse a saját vezérlők készítésének lehetőségeit!
 - > Ismertesse a UserControl felhasználásának lehetőségeit!
- A destruktur, Finalize és a Dispose
 - > Ismertesse a Finalize metódus és a destruktur kapcsolatát, valamint ismertesse működésüket!
 - > Milyen esetben írunk .NET környezetben destruktort?
 - > Ismertesse a Dispose minta lényegét! Mi a dispose alkalmazásának célja?
 - > Mutasson példát egy az IDisposable interfész implementáló osztály helyes használatára!

Irodalom

- Getting Started with Windows Forms:
 - > <http://msdn.microsoft.com/en-us/library/ms229601.aspx>
- Windows Forms programming:
 - > <http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx>
- Implementing a Dispose Method
 - > <http://msdn2.microsoft.com/en-us/library/fs2xkftw.aspx>
- Graphics and Drawing in Windows Forms
 - > <http://msdn.microsoft.com/en-us/library/a36fascx.aspx>
- Graphics Overview (Windows Forms)
 - > <http://msdn.microsoft.com/en-us/library/haxsc50a.aspx>

Szoftvertechnikák

Folyamatok (processzek),
Konkurens alkalmazások fejlesztése



Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

Alapok

- > Folyamat (Process)
- > Szál (Thread)

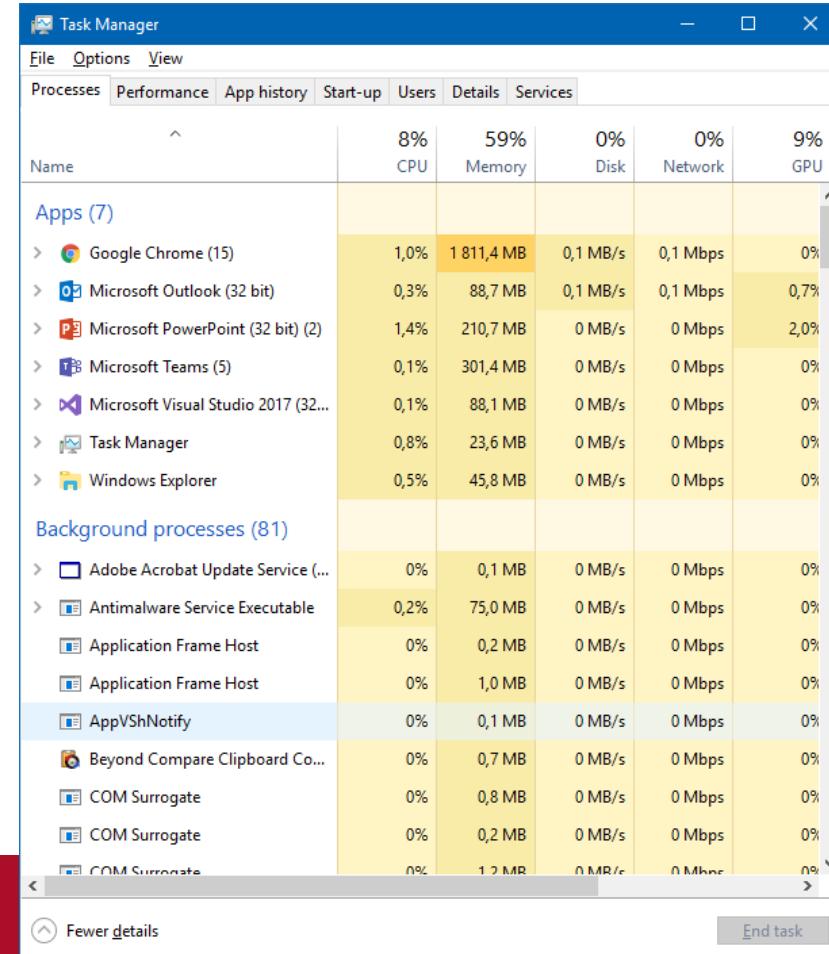
Szálkezelés

Szálak/folyamatok
szinkronizálása

Folyamat

Folyamat (process, processz)

- Alkalmazás (program) \neq folyamat
 - > A folyamat a program egy betöltött példánya
- Mi tartozik a folyamathoz
 - > Egy saját **címtartomány** (private address space)
 - > Rendszererőforrások
 - > Legalább egy szál (thread) fut
- Minek az egysége a folyamat?



The screenshot shows the Windows Task Manager window with the "Processes" tab selected. The table displays various running applications and system processes across six columns: Name, CPU, Memory, Disk, Network, and GPU. The "CPU" column is currently sorted in descending order.

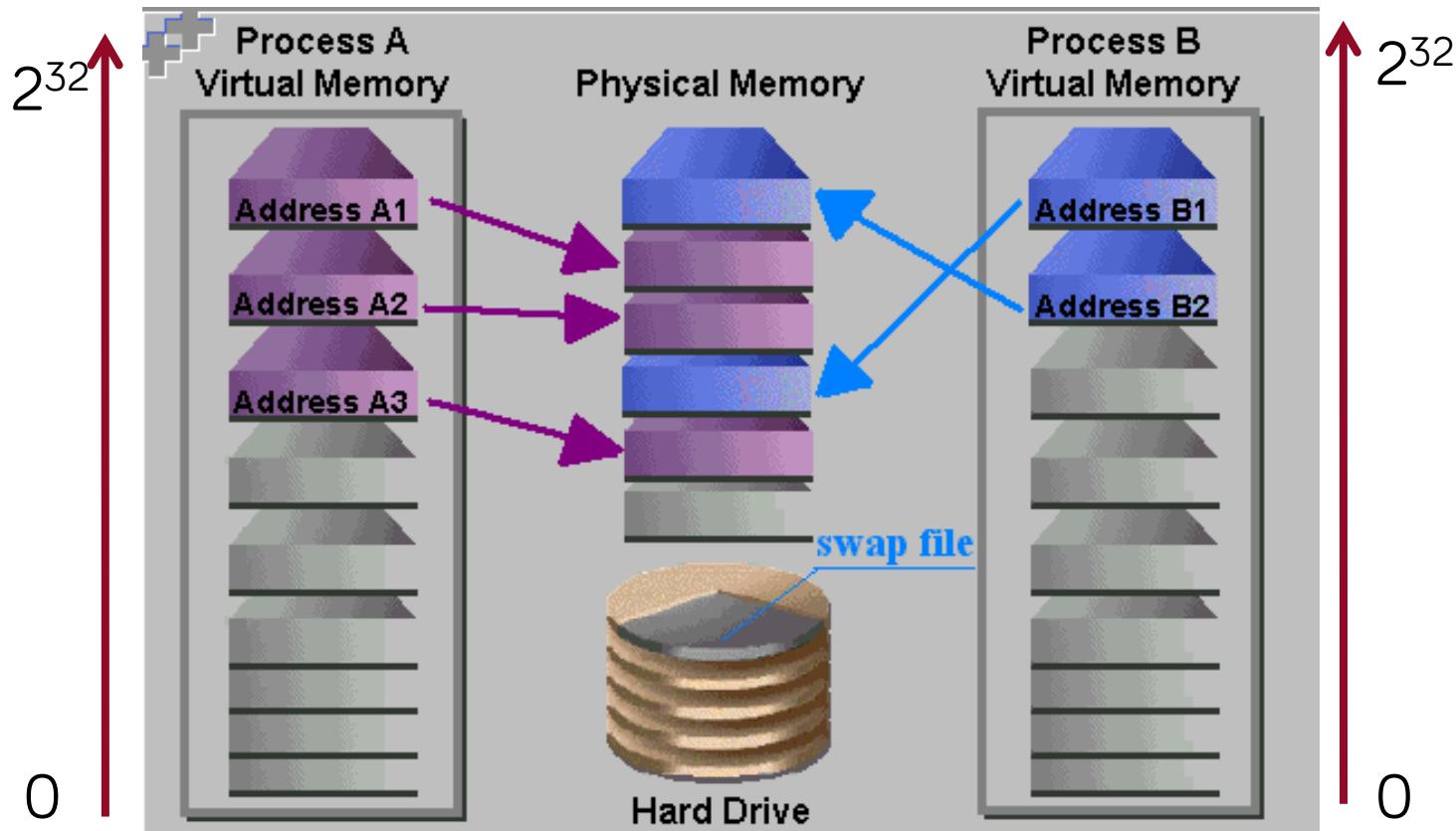
Name	8% CPU	59% Memory	0% Disk	0% Network	9% GPU
Apps (7)					
Google Chrome (15)	1,0%	1 811,4 MB	0,1 MB/s	0,1 Mbps	0%
Microsoft Outlook (32 bit)	0,3%	88,7 MB	0,1 MB/s	0,1 Mbps	0,7%
Microsoft PowerPoint (32 bit) (2)	1,4%	210,7 MB	0 MB/s	0 Mbps	2,0%
Microsoft Teams (5)	0,1%	301,4 MB	0 MB/s	0 Mbps	0%
Microsoft Visual Studio 2017 (32...)	0,1%	88,1 MB	0 MB/s	0 Mbps	0%
Task Manager	0,8%	23,6 MB	0 MB/s	0 Mbps	0%
Windows Explorer	0,5%	45,8 MB	0 MB/s	0 Mbps	0%
Background processes (81)					
Adobe Acrobat Update Service (...)	0%	0,1 MB	0 MB/s	0 Mbps	0%
Antimalware Service Executable	0,2%	75,0 MB	0 MB/s	0 Mbps	0%
Application Frame Host	0%	0,2 MB	0 MB/s	0 Mbps	0%
Application Frame Host	0%	1,0 MB	0 MB/s	0 Mbps	0%
AppVShNotify	0%	0,1 MB	0 MB/s	0 Mbps	0%
Beyond Compare Clipboard Co...	0%	0,7 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	0,8 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	0,2 MB	0 MB/s	0 Mbps	0%
COM Surrogate	0%	1,2 MB	0 MB/s	0 Mbps	0%

Folyamat

- A folyamat alapvetően **VÉDELMI** egység!
 - > A futási/ütemezési egység a **szál**
- A folyamatok egymástól **elszigeteltek**
 - > Védettek egymástól
 - > Az OS védett a felhasználói folyamtoktól
 - > Egy folyamat megszűntetésének nincs hatása a többire/OS-re
- Mi védett? A memória!
 - > minden folyamat saját kb. 2 GB virtuális címtartományt kap (32 bites OS esetén)
 - A teljes címtartomány egy 32 bites OS alatt 2^{32} (4 GB), de az OS kb. 2 GB-ot fenntart magának
 - > A virtuális címeket a memóriamenedzser fizikai címekre képezi le (RAM)

Memóriakezelés

- A leképezés során a fizikai memóriában más címekre képződnek le a folyamatok virtuális címei



Memóriakezelés

- 32 bites OS alatt minden folyamat kb. 2GB virtuális címtartománnyal rendelkezik
- Nem áll automatikusan rendelkezésre: **használat előtt foglalni kell!**
 - > C/C++ globális/statikus változók – A folyamat indulásakor allokálódik hely
 - > Lokális változók – A Stacken allokálódik hely
 - > malloc, new – Dinamikusan allokálunk helyet
- A színfalak mögött az **OS-től történik memóriakérés minden esetben**. API függvények pl.:
 - > VirtualAlloc, HeapAlloc, ...
- Ha érvénytelen címre hivatkozunk:
 - > „Access violation at address ...” Üzenet

```
int* p = 0xfa2343c2;  
*p = 21;
```

Folyamatok „programozása”

- API
 - > CreateProcess(...) – elindít egy folyamatot, többek között egy futtatható fájl útvonala a paraméter
 - > Stb.
- .NET
 - > System.Diagnostics.Process osztály
 - Start() – elindít egy folyamatot
 - Kill() – terminálja a folyamatot
 - Process.GetCurrentProcess() – aktuális process elérése
 - > Mit csinál a következő kódrészlet?

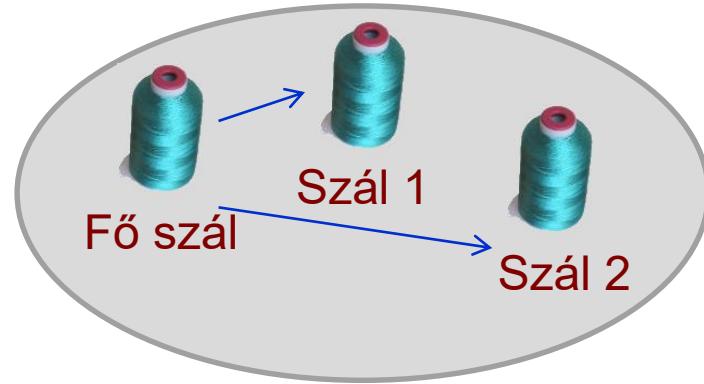
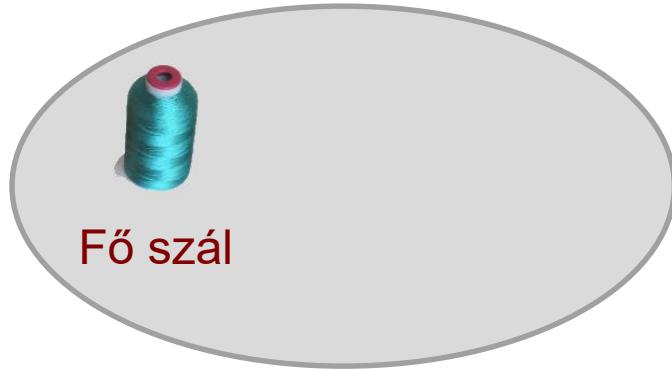
```
Process.GetCurrentProcess().Kill();
```



Többszálú alkalmazások

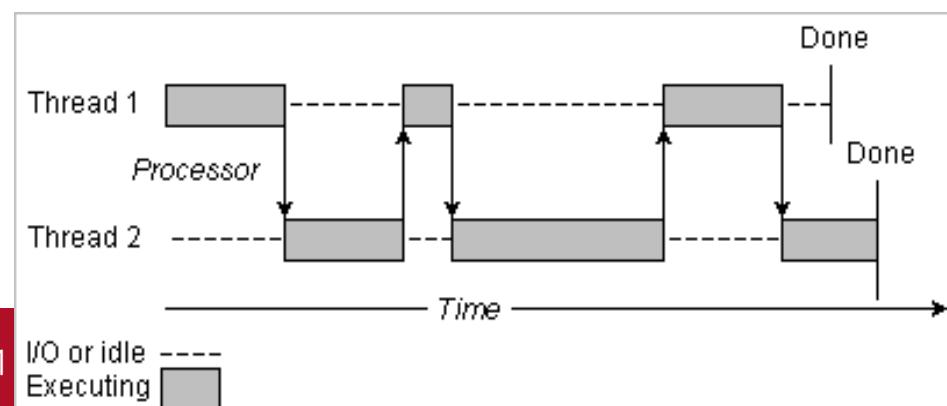
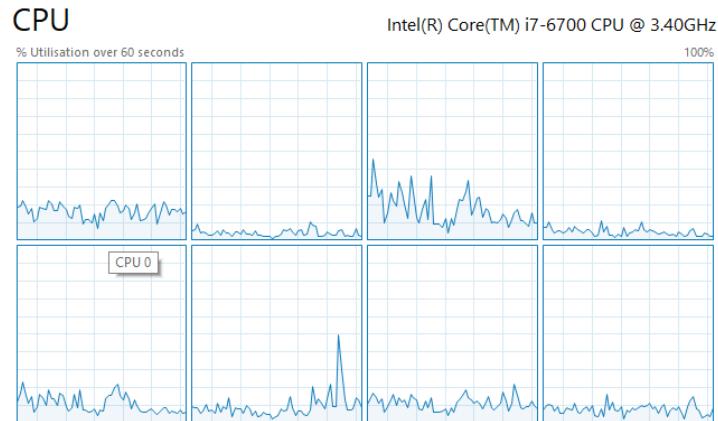
A szál fogalma

- A szál (thread) a futási/ütemezési egység
- Az eddigi alkalmazásaink **egyszálúak** voltak
 - > A folyamat elindulásakor létrejött a folyamat **fő szála** (main thread)
 - > Ebből indíthatunk újabb szálakat, ekkor az alkalmazásunk **többszálú** (multithreaded) lesz.
 - > A szál egy folyamaton belüli feladatként fogható fel



Szálak ütemezése

- Az OS a szálakat ütemezi
 - > Kiválaszt egyet a futásra kész szálak közül, és a processzort hozzá rendeli
 - > Egy CPU-n egyidőben egy szál futhat (a processzormagok száma számít)
- Ütemezési típusok
 - > Nem preemptív: egy szál csak akkor kaphat futási jogot, ha önként lemond futási jogáról. Ilyen pl. a Windows 3.1. Kiéheztetés!
 - > Preemptív: az ütemező egy idő után elveszi a futási jogot a száltól, ha nagyobb prioritású szál futásra kész, vagy lejárt a szál időszelete ($n * 10$ millisec). A legtöbb modern OS ilyen. Egy CPU esetén is látszólag párhuzamosan futnak a szálak.
- Szálak állapota
 - > Futó
 - > Felfüggesztett
 - > I/O műveletre vár
- Szálak prioritása
 - > Visszatérünk ...



Többszálú alkalmazások előnyei

- 1. Átlagosan jobb CPU kihasználás elérése
 - > Amíg egy szál IO műveletre vár
 - (pl. adat olvasása diszkről), addig más szál futhat
 - > Több CPU (CPU mag) kihasználása egy alkalmazáson belül
 - Egy szál csak egy CPU-n tud futni. Ha egy alkalmazáson belül szeretnénk több CPU magot is kihasználni, az csak akkor lehetséges, ha az alkalmazás több szálat indít.
 - Pl. egy adathalmaz feldolgozását több szálra bízzuk, minden egyik szál egy adott rész feldolgozásáért felel.

DEMO

Többszálú alkalmazások előnyei

- 2. Hosszú blokkoló művelet GUI alkalmazásokban
 - > Ne futtassuk a fő szálban, mert a GUI befagy
 - > A művelet háttérszálban futtatásával a főszál reaktív marad. A főszál feladata ez esetben a felhasználói események kezelése.

DEMO

Többszálú alkalmazások előnyei

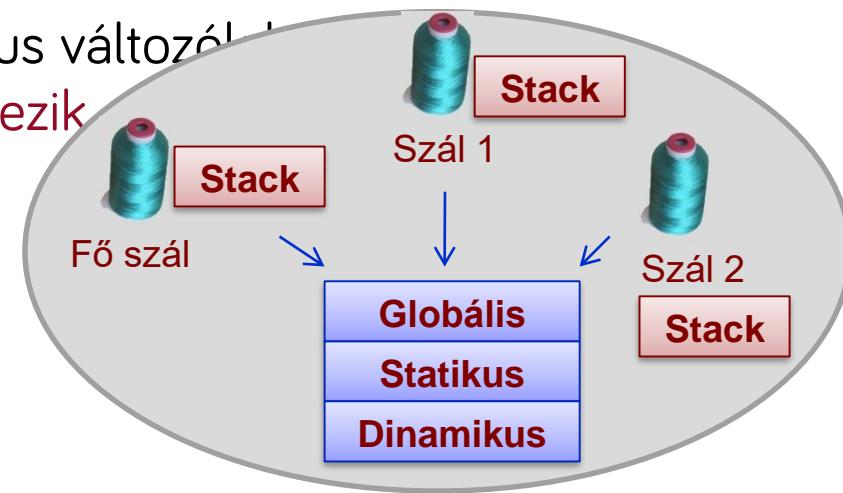
- 3. Időzítésérzékeny feladatok
 - > Pl. folyamatvezérlés külön, nagyprioritású szálban futtatása.
- 4. Kiszolgáló (szerver) alkalmazások (pl. webszerver) esetében kisebb átlagos válaszidő
 - > Ha a kiszolgáló alkalmazás egyszálú, akkor egy olyan kliens kiszolgálása, mely sokáig tart, hosszú ideig feltartja az ezzel egyidőben beérkező, de elvileg gyorsan kiszolgálható kliensek kéréseit (mert a kérések kiszolgálása az egyszálúság miatt sorban történik).
 - > Ha a kiszolgáló többszálú (az egyes klienseket más-más szál szolgálja ki), az időszemeletes ütemezés miatt még egy CPU esetén sem áll fent az előző pontban vázolt „kiéheztetés” esete.

Szál ↔ Folyamat

- A szál sokkal kisebb erőforrásigényű, mint a folyamat
 - > Különösen Windows környezetben, Linux/Unix alatt ez kevésbé igaz
- A **folyamatok** elszigeteltek: a folyamatok közötti kommunikáció nehéz
- A **szálak** egy adott folyamaton belül egy címtartományban vannak, így „könnyen” kommunikálnak
 - > A C++ a globális, statikus, dinamikus változókat használja
 - > minden szál saját stackkel rendelkezik
 - A szál lokális változóit csak az adott szál látja



Folyamat 1



Folyamat 2

Szinkron/aszinkron végrehajtás

- Definíciók
 - > **Szinkron végrehajtás:** A hívó a művelet befejezéséig várakozik (blokkolt)
 - > **Aszinkron végrehajtás:** A hívó a művelet befejezését nem várja meg. A rendszer a hívót valamilyen módon értesíti a művelet befejezéséről (eredmény)
- Aszinkron végrehajtás hogyan valósítható meg?
 - > I/O műveletek (fájlkezelés, hálózatkezelés, stb.) esetén **aszinkron I/O végrehajtással** (ha az OS támogatja). Az OS belső megszakításkezelésére épül, nem igényel külön szálat.
 - > Megoldható a **művelet külön szálban futtatásával**. Ez költségesebb, külön szálat igényel, de nem csak I/O esetén használható.

Natív szálkezelés

- Win32 API szálkezelő függvények
 - > Lényeges
 - CreateThread – elindít egy szálat
 - > További függvények (nem kell tudni)
 - **TerminateThread** – terminál egy szálat, SOHA NE HASZNÁLJUK
 - SuspendThread – szál felfüggesztése
 - ResumeThread – kiléptetés felfüggesztett állapotból
 - Sleep – szál elaltatása adott ideig
 - ...



A szálkezelés alapjai .NET felügyelt környezetben

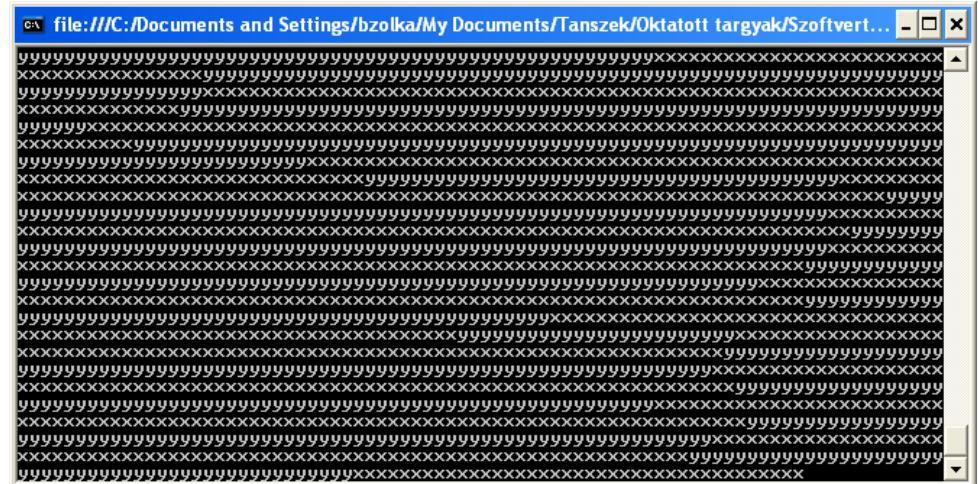
Szál indítása

- System.Threading névtér,
- Thread osztály
- Indítsunk szálat ➔

Szál indítása

```
class Program
{
    static void Main(string[] args)
    {
        Thread t = new Thread(new ThreadStart(WriteY));
        t.Start();
        while (true)
            Console.Write("x");
    }

    static void WriteY()
    {
        while (true)
            Console.Write("y");
    }
}
```



- A WriteY az ún. szálfüggvény.

Szál indítása

- ThreadStart delegate

```
Thread t = new Thread(new ThreadStart(WriteY));
```

- Egyszerűbb forma .NET 2.0-tól

```
Thread t = new Thread(WriteY);
```

➢ .NET 2.0-tól a delegate-ek esetén ez általánosságában megtehető.

- A Start() művelet indítja a szálat

➢ Hívásáig a szálat az OS nem ütemezi

- Ha kilép a szálfüggvény: a szál befejezi futását

- Paraméter is átadható a szálfüggvénynek

➢ ParameterizedThreadStart delegate

Paraméterezett szálindítás

```
class Program
{
    static void Main(string[] args)
    {
        Thread t = new Thread(new ParameterizedThreadStart(WriteAny));
        // Így is OK lenne.
        // Thread t = new Thread( WriteAny );
        t.Start("Z"); // paraméter a szálfüggvénynek
        while (true)
            Console.Write("X");
    }
    static void WriteAny(object param)
    {
        while (true)
            Console.Write(param);
    }
}
```

Adat átadása szálnak objektum metódusreferenciával

- Objektum (nem statikus) metódusreferencia átadásával

```
class ThreadClass
{
    private string text;

    public ThreadClass(string text)
    {
        this.text = text;
    }

    public void WriteAny()
    {
        while (true)
            Console.Write(text);
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        ThreadClass ts =
            new ThreadClass("Z");
        Thread t =
            new Thread(ts.WriteAny);
        t.Start();
        while (true)
            Console.Write("X");
    }
}
```

Előtér- és háttérszálak

DEMO2

- A létrehozott szál alapértelmezésben előtér szál
- Egy processz csak akkor lép ki, ha minden előtér szál befejezte a futását.
- Háttérszál indítása

```
static void Main(string[] args)
{
    Thread t = new Thread(ThreadFunc);
    t.Start();
    t.IsBackground = true;
    Console.WriteLine("Én vagyok a főszál és végezem...\n");
}

static void ThreadFunc()
{
    for (int i = 0; i < 5; i++)
    {
        Thread.Sleep(1000);
        Console.WriteLine("Én vagyok a háttérszál: " + i);
    }
}
```

Néhány hasznos művelet/tulajdonság

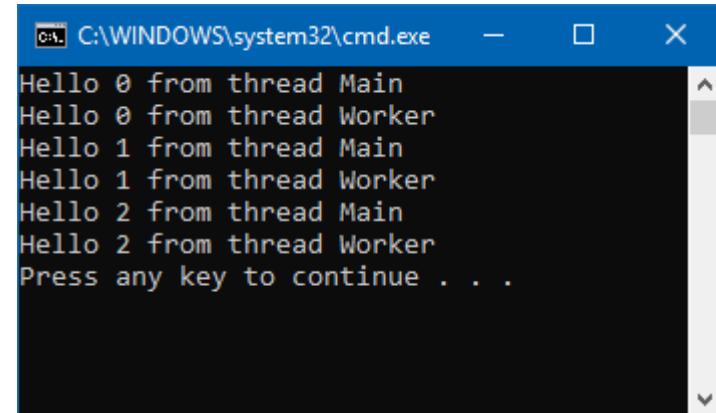
- Thread.CurrentThread statikus property
 - > Aktuális szálat adja vissza
- Name tulajdonság
 - > Név adható a szálnak (nyomkövetést segíti)
- Thread.Sleep(int millisec) és Thread.Sleep(TimeSpan ts) statikus művelet
 - > Elaltatja a hívó szálat
 - > Bem foglal CPU időt

Példa

DEMO3

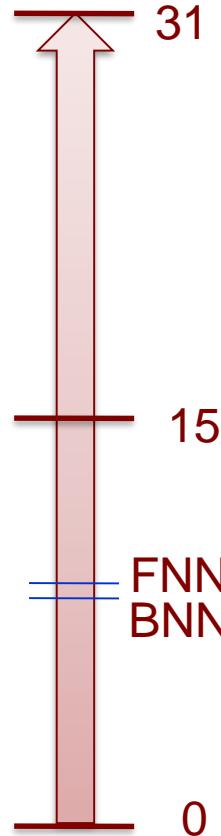
```
class Program
{
    static void Main()
    {
        Thread.CurrentThread.Name = "Main";
        Thread worker = new Thread(SayHello);
        worker.Name = "Worker";
        worker.Start();
        SayHello();
    }

    static void SayHello()
    {
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine("Hello {0} from thread {1}",
                i, Thread.CurrentThread.Name);
            Thread.Sleep(1000);
        }
    }
}
```



Szálak prioritása

- Az effektív prioritást folyamat prioritási osztálya és a szál prioritása együttesen határozza meg
- Folyamatprioritás
 - > enum ProcessPriorityClass { Idle, BelowNormal, **Normal**, AboveNormal, High, RealTime}
- Szálprioritás
 - > enum ThreadPriority { Lowest, BelowNormal, **Normal**, AboveNormal, Highest }
- Az alapértelmezett mindkettőre a Normal
- OS függő a pontos működés
- Windows Forms alkalmazásoknál az aktív ablak szála kap(hat) egy kis boostot



```
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;  
Thread.CurrentThread.Priority = ThreadPriority.Highest;
```

Szálak ütemezése – nem csak .NET

- **RealTime** folyamatprioritás
 - > Ezzel arra kérjük az OS-t, hogy soha ne vegye el a futási jogot.
 - > Veszélyes játék, kizárhatjuk vele az OS-t!
 - Ha végtelen ciklusba kerülünk, csak a reset gomb segít!
 - Ha nem adunk futást az OS-nek, „befagy” az egér, a harddisk nem tud írni, stb.
 - > Az OS egy alacsonyabb szál prioritását megnövelheti, ha egy nagyobb prioritású szál vár az eredményére.
- **Ha „garantált” válaszidő szükséges**
 - > (pl. folyamatvezérlés, szimuláció, hangátvitel (Skype), multimédia lejátszás, stb.), célszerű különválasztani a következőket
 - > Folyamatvezérlés: nagy /realtime prioritású szálba/folyamatba
 - > GUI megjelenítés: kisebb prioritású szálba/folyamatba
 - A GUI újrafestése így nem használja el a CPU erőforrás nagy részét
 - Ha a beavatkozás kritikus (pl. nyomógombra leállítás), akkor ez nem tehető meg, vagy tegyük a beavatkozást lehetővé tevő felületeket egy harmadik nagyprioritású szálba/folyamatba.

„Realtime” kitérő

- Valósidejű (realtime) Operációs Rendszerek
 - > **Hard realtime**
 - Garantált válaszidő/végrehajtásidő adott t időn belül. Nem feltétlenül jelenti azt, hogy gyorsan, t lehet „nagy”!
 - Pl. ha emberélet függhet rajta
 - > **Soft realtime:**
 - A válaszidő nagy valószínűséggel biztosított.
- Megjegyzések
 - > A szemétgyűjtést alkalmazó környezetek általánosságában kevésbé determinisztikusak.
 - Szimulációt célszerűbb C++-ban írni
 - A multimédia alkalmazások hajlamosabbak akadozni, ha fut a szemétgyűjtő (.NET pl. nem, de a .NET Compact Framework igen)

Kivételkezelés

- .NET 2.0-tól: ha kezeletlen kivétel van bármilyen szálban → az alkalmazás kilép
- A kivételt a szálban kell kezelní, az indító nem kapja meg (kivétel az async delegate)!

```
public static void Main()
{
    // Eköré hiába tennénk try-catch blokkot
    new Thread(ThreadFunc).Start();
}

static void ThreadFunc()
{
    try { throw new Exception("Hiba"); }
    catch (Exception ex)
    {
        // Pl. naplózzuk a kivételt, jelezzük a hibát az indítónak stb.
    }
}
```

Kivételkezelés Windows Forms alkalmazásokban

- Az eseménykezelők kivételei egységesen kezelhetők
 - > Application.ThreadException esemény
 - > Csak az üzenetkezelők kivételeit kezeli: a worker szálak kivételeit, a main fv konstruktörának kivételeit
(ami a message loop előtt fut) nem!
 - > A kivételek ThreadException –höz irányítása kikapcsolható

```
static class Program
{
    static void Main()
    {
        Application.ThreadException += HandleError;
        Application.Run(new Form1());
    }

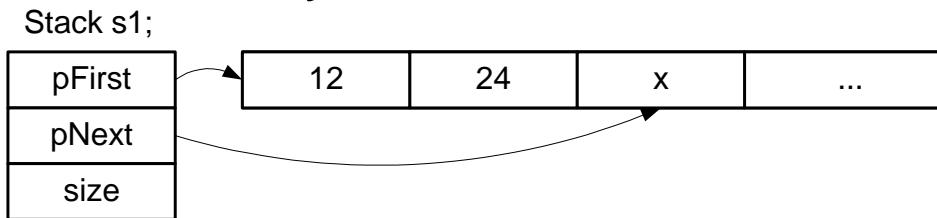
    static void HandleError(object sender, ThreadExceptionEventArgs e)
    {
        // Log exception, then either exit the app or continue...
    }
}
```

Szálak szinkronizációja

Mire kell odafigyelni a szálak kommunikációja során?

A kölcsönös kizárási problémája

- Környezet: megosztott erőforráshoz több szál fér hozzá
- Megosztott erőforrás: memória (változók, objektumok), fájl, stb.
- **Meg kell őrizni a konzisztenciát.** Egy CPU esetén sem tudjuk, mikor veszi el az ütemező a szál futási jogát!
 - > Példa: egy Stack osztály használata több szálból



pNext: első szabad rekesz

- > Példa: Person objektumok adatainak fájlba/memóriába írása több szálból
 - Előfordulhat, hogy féligríttük csak ki az adott objektumot és egy másik író szál kapja meg a futási jogot:



A kölcsönös kizárási problémája

- Hogyan őrizzük meg? Kölcsönös kizárással
 - > Biztosítanunk kell, hogy a megosztott erőforráshoz egy időben csak egy szál férjen hozzá.
- Próbálkozzunk: vezessünk be egy **bool flag** változót, ami jelzi, hogy foglalt-e az erőforrás (ha foglalt, várni kell).

```
while (flag) ; // Várunk, ha foglalt  
flag = 1; // Először foglaljuk  
// Itt használjuk az erőforrást  
flag = 0; // Ha már nem használjuk
```

Szál A

```
while (flag) ;  
flag = 1;  
// Itt használjuk az erőforrást  
flag = 0;
```

Szál B

- Ha pont a while(flag); és a flag = 1; között veszti el SzálA a futási jogát: baj van → minden két szál szabadnak látja!

Kölcsönös kizárási mechanizmus

- A megoldás: ún. oszthatatlan test_and_set utasításra van szükség.
 - > Ezt a hardver architektúrának kell támogatnia
 - > Erre építve megoldható a kölcsönös kizárási problémája, **kritikus szakasz** alakítható ki és további szinkronizációs konstrukciók készíthetők
- **Kritikus szakasz**
 - > Az a kód részlet, amelyből a megosztott erőforráshoz férünk hozzá, vagyis amelyre garantálni kell az atomi/oszthatatlan hozzáférést
 - > Az OS és a .NET támogatja kialakítását, rövidesen látjuk
- További probléma volt
 - > A `while(flag);` teljesen fölöslegesen használja a CPU-t (akár 100%-osan is!)

.NET szinkronizációs konstrukciók I.

- A kölcsönös kizárás csak egyfajta szinkronizációs probléma...
- Zárolási konstrukciók:

Név	Cél	Folyamatok között is?	Sebesség
lock C# utasítás (Monitor.Enter/Monitor.Leave)	Biztosítja, hogy egy adott erőforráshoz/kódrészlethez egy időben csak egy szál férhet hozzá.	nem	gyors
Mutex	Mint a lock, de folyamatok között is . Pl. annak megoldására, hogy egy alkalmazásból csak egy példány indulhasson.	igen	közepes
Semaphore	Mint a Mutex, de nem egy, hanem max. N hozzáférést engedélyez.	igen	közepes
ReaderWriterLock	Sok olvasóra optimalizált megoldás. Egyszerre több olvasó is hozzáférhet az erőforráshoz, de íróból csak egy (illetve az író kizára az olvasókat is). Pl. ritkán módosított cache megvalósítása.	nem	közepes

A lock használata

- Írjuk ki csak egyszer a „Done” szöveget. Több szál „versenyez”.

```
class ThreadSafeClass
{
    static bool done; // Közös erőforrás
    static object syncObject = new object();
    static void Main()
    {
        new Thread(Run).Start();
        new Thread(Run).Start();
    }

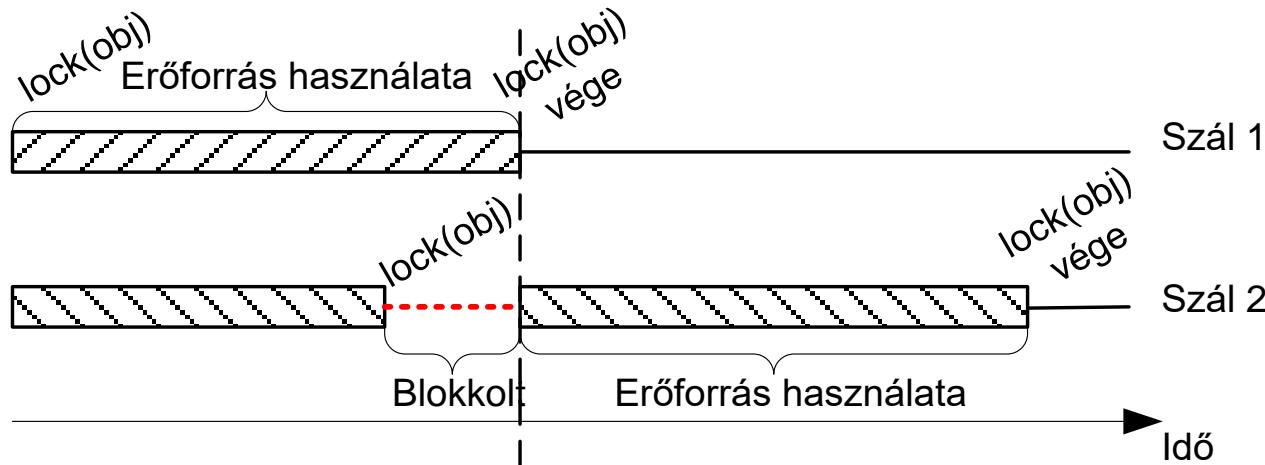
    static void Run()
    {
        lock (syncObject)
        {
            if (!done) { done = true; Console.WriteLine("Done"); }
        }
    }
}
```



A lock használata

- A lock működése

- > Egy objektum paramétert kell neki adni
- > Megvizsgálja, hogy zárolva van-e az objektum
 - Ha nincs, atomi módon zárolja, majd tovább fut (a lock tulajdonképpen egy oszthatatlan test_and_set)
 - Ha igen, vár (blokkolja a hívó szálat), amíg fel nem szabadul
- > A várakozás nem használ CPU időt
- > A várakozó szálak egy sorba kerülnek, „érkezési sorrendben” kapnak hozzáférési jogot
- > A lock blokkból kilépéskor oldja az objektumon levő zárat

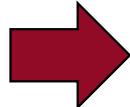


```
while (flag) ;  
flag = 1;  
// Itt használjuk az erőforrást  
flag = 0;
```

A lock és a Monitor osztály

- A lock utasításból a fordító ezt generálja

```
lock (syncObject)
{
    // ...
}
```



```
try
{
    Monitor.Enter(syncObject);
    // ...
}
finally
{
    Monitor.Exit(syncObject);
}
```

- Monitor.TryEnter – hasonló, de időkorlát is megadható

Zárak egymásba ágyazása

- A zároló szál ismételten hívhat lock-ot, nem kerül blokkolásra.
- A zár elengedéséhez a zárolással megegyező számú zárfeloldás szükséges

```
static object syncObject = new object();

static void Main()
{
    lock (syncObject)
    {
        Console.WriteLine("lockolva");
        Nest();
        Console.WriteLine("még mindig lockolva");
    } // Lock elengedése
}

static void Nest()
{
    lock (syncObject)
    {
        // ...
    } // Marad a lock
}
```

Mi lehet szinkronizációs objektum?

- Vagyis mi lehet lock() paraméter?
 - > Csak referencia típus (osztály)!
- Hogyan védjük a tagváltozókat?
 - > A nem statikusakat nem statikus tagváltozóval (objektumszintű zár)
 - > A statikus tagváltozókat statikus tagváltozóval (osztályszintű zár)

Objektumszintű:

```
class ThreadSafeClass
{
    long var = 0;
    object syncObject = new object();

    void Increment()
    {
        lock (syncObject) { var++; }
    }
}
```

Osztályszintű:

```
class ThreadSafeClass
{
    static long staticVar = 0;
    static object classSyncObject
        = new object();

    static void IncrementStatic()
    {
        lock (classSyncObject)
        { staticVar++; }
    }
}
```

Szálbiztos (thread-safe) osztályok

- Egy osztály szálbiztos, ha úgy lett megírva, hogy többszálú környezetben is biztonságosan használható.
 - > Maga garantálja a konzisztenciát
 - > A felhasználás során már nem kell zárolni...
- Szálbiztos Stack osztály ➔
 - > (üres/tele feltétel ellenőrzések nélkül)

```
public class Stack<T>
{
    readonly int size;
    int current = 0;
    T[] items;
    object syncObject = new object();

    public void Push(T item)
    {
        lock (syncObject)
        {
            items[current++] = item;
        }
    }

    public T Pop()
    {
        lock (syncObject)
        {
            return items[--current];
        }
    }
}
```

Szálbiztos osztályok

- Tegyünk minden osztályt szálbiztossá?
 - > A zárolás viszonylag időigényes ...
 - > Ezért ha az osztályt az esetek többségében egyszálú környezetben használják, akkor nem célszerű.
 - Ekkor is lehet köré egy szálbiztos csomagolóosztályt (wrapper) írni, aminek az interfésze ráadásul megegyezhet az eredeti osztállyal.
- .NET keretrendszer osztályok
 - > Tipikusan csak a statikus tagváltozók védettek, így csak a statikus műveletek/tulajdonságok szálbiztosak (nézzük meg a dokumentációt)
- Kitérő - C környezetben (pl. a printf függvény szálbiztos?)
 - > A globális és a statikus lokális változók a problémásak
 - > Pl. a C Runtime Librarynek van szálbiztos és nem szálbiztos változata, linker beállítás függő melyiket használjuk

Mi szálbiztos és mi nem?

- Mit kell védeni?
 - > A C# nyelven csak a 32 bites és az annál kisebb változók olvasása, írása atomi

```
static int x, y;
static long z;

static void Test()
{
    long myLocal;
    x = 3; // Atomi
    z = 3; // Nem atomi 32 bites OS alatt (z 64 bites)
    myLocal = z; // Nem atomi
    y += x; // Nem atomi: két művelet (olvasás + írás)
    x++; // Nem atomi: két művelet (olvasás + írás)
}
```

- Hogyan védhető
 - > Lock-kal/Mutex-szel vagy:
 - > Nemblokkoló atomi utasítással: **InterlockedXXX**, nagyon hatékony →

Interlocked atomi utasítások

- Mit kell tudni? Egy példát mutatni (pl. kiemelt rész).

```
class Program
{
    static long sum;

    static void Main()
    {
        Interlocked.Increment(ref sum);
        Interlocked.Decrement(ref sum);
        Interlocked.Add(ref sum, 3); // Érték hozzáadása
        // 64 bites érték olvasása
        Console.WriteLine(Interlocked.Read(ref sum));
        // Érték írása és az előző olvasása atomi módon
        // Ez 3-at ír ki, sum 10 lesz
        Console.WriteLine(Interlocked.Exchange(ref sum, 10));
        // Ha sum 10, akkor 123-at ír bele atomi módon
        Interlocked.CompareExchange(ref sum, 123, 10);
    }
}
```

Volatile mezők

```
class Foo  
{  
    public int x;  
    public int y;  
}
```

Nem biztos,
hogy 5-öt ír ki !

```
// szál 1  
foo.y = 5;  
foo.x = 1;
```

```
// szál 2  
if (foo.x == 1)  
{  
    Console.WriteLine(foo.y);  
}
```

- Előfordulhat, hogy az y változó értéke regiszterben chache-elt, ezt a szál 2 nem látja
- A compiler számára megengedett, hogy a nem-volatile mezők írási és olvasási sorrendjét felcserélje, ha egy szalon nem látszik a különbség

Volatile mezők

- Megoldás
 - > A volatile írás nem lehető későbbre.
 - > A volatile olvasás nem lehető előbbie.
 - > „Azonnal” megtörténik a memóriába írás (nem cache-elt regiszterben)

```
class Foo
{
    public volatile int x;
    public volatile int y;
}
```

- Megjegyzés: lock esetén nincs szükség a volatile alkalmazására
 - > lock blokkba belépéskor minden megtörténik a változók memóriából frissítése
 - > lock blokkból kilépéskor minden megtörténik a változók memóriába kiírása

Szál kiléptetése – Alternatíva 1

- Egy bool változóval
 - > A szálnak rendszeres időközönként rá kell nézni!
- Szál kilépésének bevárása: Thread.Join()

```
class ThreadClass
{
    static volatile bool exit = false;

    static void Main()
    {
        Thread thread = new Thread(Run);
        thread.Start();
        Thread.Sleep(2000);
        Console.WriteLine(
            "Signaling to worker thread.");
        exit = true;
        Console.WriteLine("Waiting for
            worker thread to exit.");
        thread.Join();
        Console.WriteLine("Worker thread
            definitely has exited.");
    }
}
```

```
static void Run()
{
    int counter = 0;
    while (!exit)
    {
        Thread.Sleep(300);
        Console.WriteLine(
            counter++.ToString());
    }

    Console.WriteLine(
        "Exiting from worker thread.");
}
```

.NET szinkronizációs konstrukciók II.

- A **lock** megoldja a kölcsönös kizárás problémáját
- Más szálnak jelzésre (pl. munka kezdhető, lépj ki, stb.) nem alkalmas

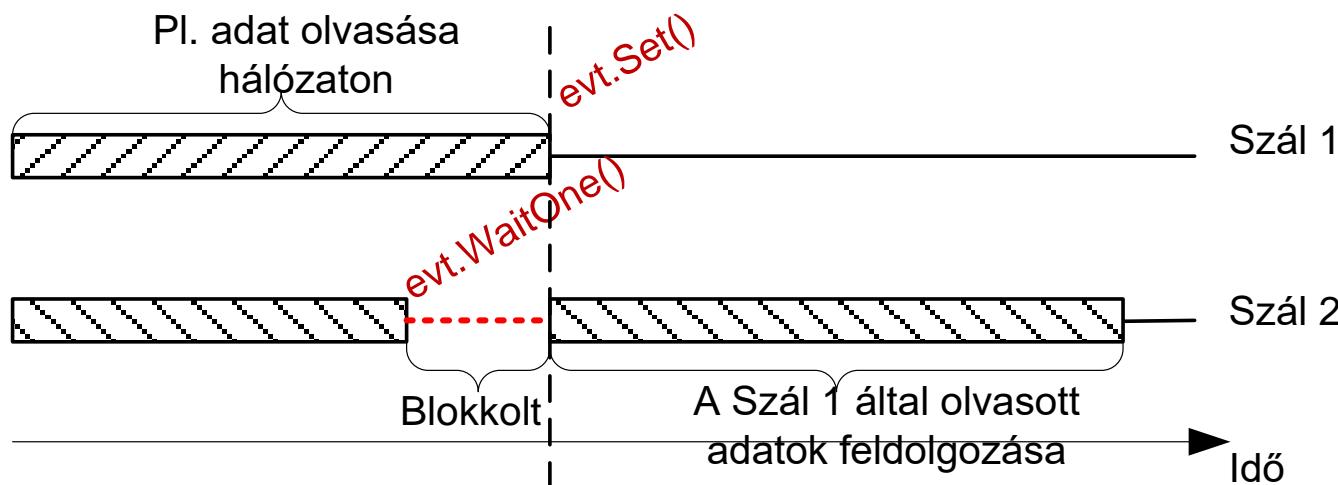
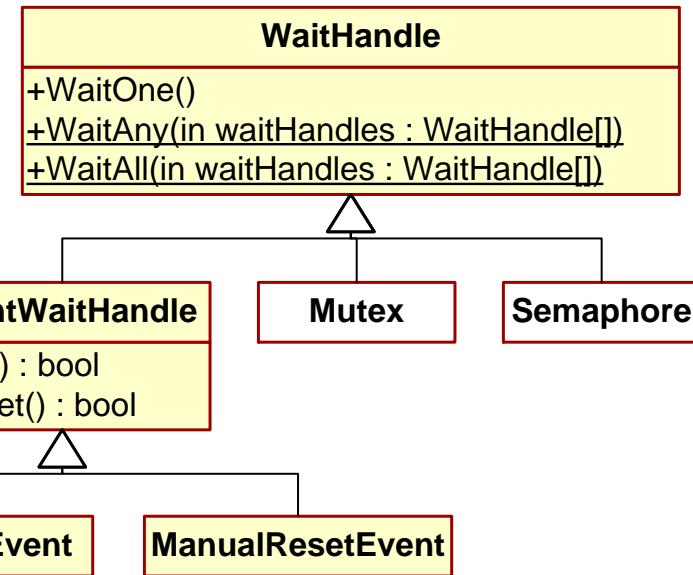
Jelzésre alkalmas szinkronizációs konstrukciók:

Név	Cél	Folyamatok között is?	Sebesség
EventWaitHandle	Lehetővé teszi hogy egy szál hatékony módon várakozzon egy más szál által küldött jelzésre.	igen	közepes
Monitor.Wait and Monitor.Pulse	Lehetővé teszi hogy egy szál hatékony módon várakozzon tetszőleges blokkoló feltételre.	nem	közepes

WaitHandle hierarchia

- **WaitHandle**

- > Olyan osztályok őse amelyek objektumaira várakozni lehet
- > Az AutoResetEvent és ManualResetEvent eseményre való várakozást tesz lehetővé



`AutoResetEvent evt = new AutoResetEvent(false);`

AutoResetEvent

- Olyan, mint a sífelfonó beléptető...
- **Set** művelet:
 - > Jelzettbe állítja az event objektumot, így elenged egy várakozó objektumot.
 - > Ha nincs várakozó objektum, jelzett marad.
 - > Azt nem tartja nyilván, hányszor volt Set hívás (a több is egynek számít).
- **WaitOne** művelet:
 - > Ha az event objektum nem jelzett, várakozik (a CPU-t nem terheli)
 - > Ha az event objektum jelzett:
 - Tovább fut
 - AutoResetEvent esetén automatikusan nem jelzettbe állítja az event objektumot (ManualResetEvent esetén nem)
- **Reset** művelet:
 - > Reseteli (nem jelzett állapotba állítja) az event objektumot

AutoResetEvent példa

```
class SimpleEventDemo
{
    // A konstruktorparaméterben az event kezdeti állapotát
    // adjuk meg (false - nem jelzett).
    static EventWaitHandle wh = new AutoResetEvent(false);

    static void Main()
    {
        new Thread(Run).Start();
        Thread.Sleep(1000); // Várunk egy kicsit
        wh.Set(); // Ébresztő
    }

    static void Run()
    {
        Console.WriteLine("Várunk értesítésre ...");
        wh.WaitOne(); // Várakozás
        Console.WriteLine("Az értesítés megérkezett.");
    }
}
```

Megjegyzés: a WaitOne-nak timeout is megadható.

AutoResetEvent – példák

- AutoResetEvent – jóváhagyás példa
 - > Lásd ThreadDemo solution...
- Termelő/fogyasztó sor példa
 - > Az életben gyakran van szükség hasonlóra. Pl.:
 - Egy háttérszál olvas a soros porton/hálózatról adatokat, üzenetekké alakítja és beteszi egy sorba további feldolgozásra.
 - Terheléskiegyenlítés támogatása (burst-ösen érkeznek a feladatok, ne kelljen eldobni akkor se, ha nem tudjuk azonnal feldolgozni)
 - Skálázhatóság: több fogyasztó szál lehet, párhuzamosan (több CPU esetén) dolgozhatják fel a feladatokat
 - > Lásd ThreadDemo solution...

További WaitHandle leszármazottak

- ManualResetEvent
 - > Az AutoResetEvent „ellentéte”
 - > „Kapuként” funkcionál: ha jelzett, mindenki mehet, ha nem jelzett, mindenki vár.
- Mutex
 - > Mint a lock, kölcsönös kizáráusra
 - > Eltérő folyamatok szálai között is, használható, de a lock-nál néhány százszor lassabb.
 - > A WaitOne-nal lehet lefoglalni (ami atomi módon zárolja, illetve blokkol, ha foglalt), elengedni a ReleaseMutex-szel lehet.
- Semaphore
 - > Mint a mutex, de egynél több (N, megadható) hozzáférést enged. Ha max. N hozzáférést szeretnénk.
 - > A WaitOne-nal lehet rá várakozni. Ha még szabad a szemáfor, nem blokkol, eggyel csökkenti a szemáfor számlálót. Ha nem szabad, blokkol. Elengedni a ReleaseSemaphore-ral lehet (növeli a szemáfor számlálót).

WaitHandle

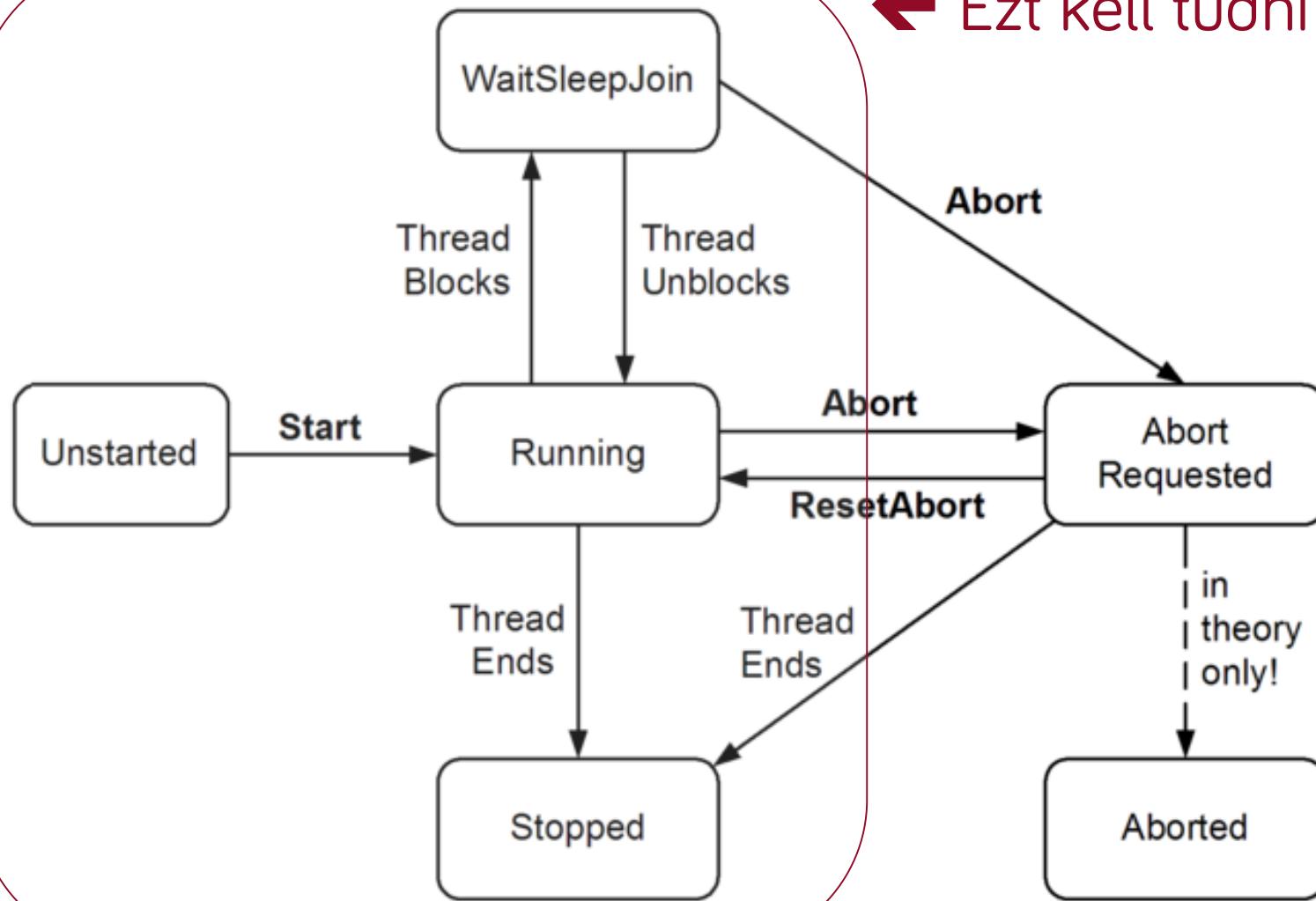
- WaitHandle leszármazott
 - > implementálja az IDisposable-t (egy natív leírót csomagol)
- **WaitHandle.WaitAny** statikus művelet
 - > Egynél több leíróra vár.
Ha bármelyik jelzett/szabad, akkor nem blokkol.
 - > Példa: lásd „Konkurens példák.doc” – „Termelő/fogyasztó sor példa gyors kiléptetéssel” (bár ez esetben felesleges két AutoResetEvent használata)
- **WaitHandle.WaitAll** statikus művelet
 - > Egynél több leíróra vár.
 - > Ha mintjelzett/szabad, csak akkor nem blokkol.
 - > Ritkán használt.

Demo

(Termelő/fogyasztó sor gyors kiléptetéssel)

Szál állapotok

← Ezt kell tudni



Szálállapotok

- **WaitSleepJoin állapot (fontos!!!)**
 - > Blokkolt állapot
 - > **Sleep, Join, lock, WaitOne, WaitAny, WaitAll** utasítások hatására
 - > Mind blokkol. Mégig? Négy módon léphet ki blokkolt állapotból:
 - A várakozási feltétel teljesül
 - Timeout lejár (ha adtunk meg)
 - A várakozás a Thread.Interrupt hívásával megszakításra kerül
 - A várakozás a Thread.Abort hívásával megszakításra kerül

Szál kiléptetése – alternatíva 2

- Eddig bool flaggel tagváltozóval. De mi van, ha osztályunk más osztályok metódusát hívja, és a blokkolás ezekben történik? Ekkor az osztályunk nem tudja a bool flaget ellenőrizni.
Demo
- A megoldás a **ThreadInterrupt** használata
 - > A szál, amire meghívjuk, kap egy **ThreadInterruptedException**-t, de csak ha **WaitSleepJoin** állapotban volt
 - Vagyis a while(true); -ből nem fogja kiléptetni a szálat
 - Ha az Interrupt hívásakor nem volt WaitSleepJoin állapotban, akkor a ThreadInterruptedException akkor keletkezik a szálban, mikor az legközelebb WaitSleepJoin állapotba lép.

Demo

(vagy következő dia)

Thread.Interrupt

```
Thread thread = new Thread(Run);
thread.Start();
Thread.Sleep(2000);
thread.Interrupt(); // Signaling to worker thread.
thread.Join(); // Waiting for worker thread to exit.
Console.WriteLine("Worker thread definitely has exited.");
// ...
static void Run()
{
    try
    {
        string task;
        while (true)
        {
            task = queue.GetTask();
        }
    }
    catch (ThreadInterruptedException)
    {
    }
}
Console.WriteLine("Exiting from worker thread.");
}
```

```
class OneItemQueue
{
    public string GetTask()
    {
        // WaitSleepJoin állapotban
        // blokkolt.
        autoResetEvent.WaitOne();
        return task;
    }
    ...
}
```

Szál kiléptetése – alternatíva 3

- `Thread.Abort()` művelet
- Hasonlít a `Thread.Interrupt`-ra, de
 - > A szál **ThreadAbortException-t** kap
 - > Nem csak `WaitSleepJoin` állapotban, hanem bármilyen állapotban lehet a szál
 - Vagyis a `while(true);` -ból is kilépteti!
 - A `finally` blokkok azért lefutnak (persze ebben lehet még egy végtelen ciklus ☺)

- Ne használjuk, mert nem tudjuk mit csinál éppen a szál! Lezáratlan dolgok maradhatnak mögöttünk...

```
StreamWriter w = File.CreateText("myfile.txt");
// Ha itt fut be az Abort, baj van!
try
{
    w.WriteLine("Abort-Safe");
}
finally { w.Dispose(); }
```



- Talán (☺) csak egy esetben:
 - > az alkalmazásból való kilépés esetén
 - > Ekkor a magunk mögött hagyott leírók úgyis felszabadulnak.

Többszálú Windows Forms alkalmazások

Architektúra

- A Windows Forms vezérlőelemek (a Control leszármazottak, így a Form is) csak abból a szálból érhetők el, ami létrehozta őket
 - > A GUI elemeket szinte mindenkor a fő szálban hozzuk létre
 - > Munkaszálból ne hívunk rájuk műveletet/property-t
 - > Csak a **Control.Invoke** hívható más szálból
 - > A **Control.Invoke** a paraméterben megadott metódusreferenciát a controlt létrehozó szálból (tipikusan a fő szálból) hívja meg.
 - > A **Control.Invoke** használatára később látunk példát (esettanulmány)
- **BackgroundWorker** osztály (nem kell tudni)
 - > A Control.Invoke hívást elrejti, magasabb absztrakciós szint
 - > Könnyű alkalmazni, használjuk a gyakorlatban!

Thread-pool

Csak gyakorlaton szerepel, de tudni kell a lényegét !!!

Thread-pool

- Kiszolgáló alkalmazást írunk. Cél több kérés párhuzamos kiszolgálása
- Megoldás 1. - minden beérkező kérésnek új szálat indítunk.
Problémák:
 - > Nagyszámú párhuzamos kérés esetén túl sok szál fut
 - Folyamatonként ~100-nál többet nem célszerű, a szálak közötti sok váltás miatt
 - > A szál indítása viszonylag költséges
- Thread-pool alkamazása az igazi megoldás
 - > Előre elindítunk néhány szálat
 - > A kiszolgálás során ezekből allokálunk
 - > A kiszolgálás végeztével a szál visszakerül a poolba, új kérést szolgálhat ki
 - > Ha nincs szabad szál a poolban:
 - Új szálat iindítunk és teszünk a poolba,
 - Ha már túl sok szál van (néhányszor tíz) - blokkoljuk a hívót míg nem szabadul fel szál

Thread-pool

- ThreadPool .NET osztály
- minden folyamattal létrejön egy
- Mikor célszerű a ThreadPool használata?
 - > Csak rövid műveleteket futtatunk.
 - > Ne blokkoljuk a ThreadPool szálakat (máskülönben hamar kimerítjük a ThreadPoolt).
- A Thread.Interrupt nem igazán használható kiléptetésre
 - > Flaggel
 - > Esetleg ManualResetEvent-tel kombinálva

A thread-pool programozása

```
class Test
{
    static ManualResetEvent doneEvent = new ManualResetEvent(false);

    public static void Main()
    {
        ThreadPool.QueueUserWorkItem(Run); // Metódusreferencia paraméter
        Console.WriteLine("Waiting for threads to complete...");
        doneEvent.WaitOne();
        Console.WriteLine("Worked has ended!");
        Console.ReadLine();
    }

    public static void Run(object instance)
    {
        Console.WriteLine("Started...");
        Thread.Sleep(1000);
        Console.WriteLine("Ended");
        doneEvent.Set();
    }
}
```

ÖSSZEFOGALÁS

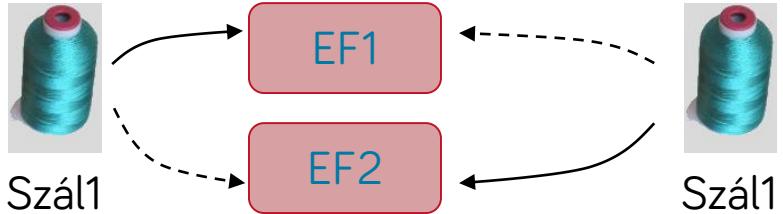
Többszálú alkalmazások hátrányai

- Növeli az alkalmazás komplexitását
 - > Meg kell oldani a szálak szinkronizációját. Pl. kölcsönös kizárás problémája.
- Nagyon-nagyon-nagyon-nagyon nehezen kinyomozható problémákat okozhat a nem megfelelő szinkronizáció
 - Pl. a kölcsönös kizárás elmaradása miatt havonta egyszer elszáll az alkalmazás
 - Nem lehet kidebuggolni, hiszen nem reprodukálható
- A túl sok szál alkalmazása terheli a rendszert
 - > Ütemezni kell, váltáskor *context switch* – CPU igény
 - > Szálanként saját stack – memóriaigény
- Holtpont (deadlock) kialakulásának veszélye
- Tanulság: a konkurens programozást nem lehet csak félíg megtanulni.

Holtpont (deadlock)

- **Holtpont:**

- > Kettő (vagy több) szál kölcsönösen egymásra vár.
- > A probléma oka: foglalva várakozás!



- **Detektálható:**

- > Megfelelő irányított gráfot kell detektálni.
- > Nekünk kell megírni.
- > Nem szoktuk.
- > Inkább megpróbáljuk elkerülni.

- **Elkerülése:**

- > Az erőforrásokat minden szálban ugyanabban a sorrendben foglaljuk le
- > Nem mindig lehető meg 😞

- **Alternatív megoldás:**

- > Adott időkorlátig várakozunk az erőforrásra (timeout alkalmazása).

Java

.NET	Java
Thread osztály	Thread osztály
Thread.Sleep, Thread.Join	Thread.Sleep, Thread.Join
lock	synchronized
AutoResetEvent, ManualResetEvent	Nincs, de hasonló hatás érhető el a Monitor.Wait, Monitor.Pulse, Monitor.PulseAll használatával.
Mutex	?
Monitor osztály	Monitor osztály

- **Megjegyzés**

- Az AutoResetEvent helyett a Monitor osztály Wait, Pulse, PulseAll művelete is használható jelzésre. A működési mechanizmus más.
- Java is támogatja
- Bár rugalmas, nehezebb megérteni, jól használni

További téma körök (nem voltak)

- Mutex, Semaphore használata
 - > Hasonlóképpen használható, mint az AutoResetEvent
- Nevesített Mutex, Event, Semaphore
 - > Ha folyamatok közötti szinkronizációra van szükség
- Monitor.Wait, Monitor.Pulse és Monitor.PulseAll használata
- BackgroundWorker osztály részletes ismertetése
- Thread Local Storage
 - > Szálhoz rendelt információ
- Magasabb szintű szálkezelés
 - > Task, Task<T>
- Aszinkronitás támogatása nyelvi szinten
 - > async, await kulcsszavak, aszinkron metódusok
- Stb.

Szoftvertechnikák

Adatbázis-kezelés alapjai, ADO.NET



Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

Relációs adatmodell
alapfogalmak

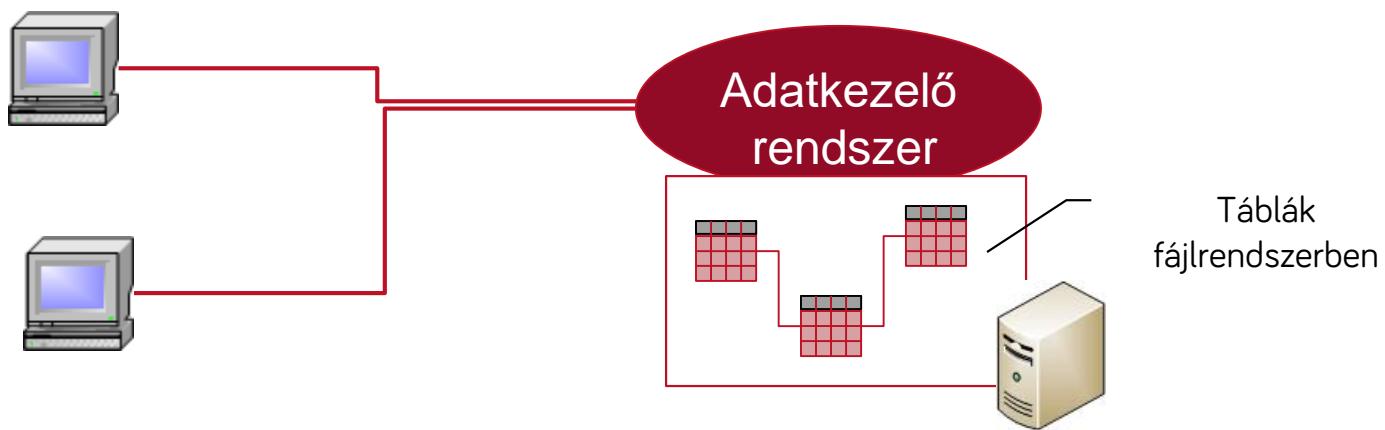
SQL

Objektumrelációs leképezés

ADO.NET

Áttekintés, architektúra

- Mit tanulunk és hogyan?
 - > Előző félévben Adatbázisok c. tárgy
 - > Jelen tárgy keretében egy rövid ismétlés a precizitás igénye nélkül
 - > Illetve lássuk, hogyan lehet egy alkalmazásból elérni egy adatbázist
- A legtöbb mai adatkezelő a **relációs adatmodellt** támogatja
 - > Nem szerver („egyfelhasználós”) termék pl.: Microsoft Access
 - > Legismertebb szerver termékek: Oracle, IBM DB2, Microsoft SQL Server, MySQL
- Egyre népszerűbbek a reláció nélküli adatbázis megoldások is (NoSQL)
 - > Általában extrém sok rekorddal rendelkező táblák esetében használják
- Architektúra (szerverek esetén, relációs adatmodellel)

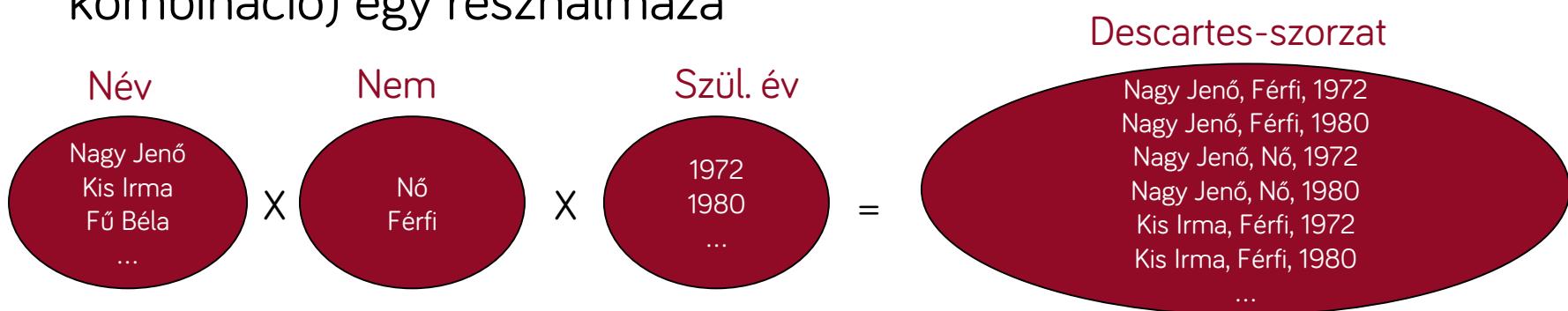


Relációs adatmodell alapfogalmak

(Adatbázisok tárgy ismétlése)

Relációs adatmodell alapfogalmak

- A legtöbb mai adatkezelő a relációs adatmodellt támogatja
- Reláció: halmazok Descartes-szorzatának (összes lehetséges kombináció) egy részhalmaza



- Vagyis a reláció egy kétdimenziós tábla, amely megnevezett oszlopokból, és korlátlan számú sorból áll. Pl.:

Név	Nem	Szül. év
Nagy Jenő	Férfi	1972
Kis Irma	Nő	1980
Fű Béla	Férfi	1938

Relációs adatmodell alapfogalmak

- Attribútum
 - > A táblázat oszlopai
 - > Az előző példában három attribútum: Név, Nem, Szül. év
 - > Adott az értelmezési tartománya (pl. bool (igaz/hamis), 32 bites szám, max 200 karakteres szöveg, stb.
- Reláció elem
 - > A táblázat egy **sora, rekordnak** is nevezük.
- Relációs séma
 - > Leírja, milyen attribútumok vannak a relációban. Jelölés:
 - RelációNeve(A1, A2, ...AN)
 - Példa: Személy(Név, Nem, Szül.Év)
 - Példa: Diák(TAJ, Törzs, Név, Lakcím, Szül)

Relációs adatmodell alapfogalmak

- Egy sor és oszlop kereszteződésében egyetlen érték szerepel
 - > minden sor egyedi, nincs két egyforma sor
 - > minden oszlopnak egyedi neve van a reláción belül
 - > az oszlopok sorrendje lényegtelen
 - > a sorok sorrendje lényegtelen
- Relációs sémák halmaza: relációs adatbázis séma
 - > összes tábla + struktúrájuk

Relációs adatmodell alapfogalmak

- Anomáliák
 - > Példa: minden hallgató egy szakirányra jár, ami egy tanszékhez tartozik.

NeptunKód	Név	Évfolyam	SzakirányNév	Tanszék	Szakir. felelős
SD3ER5	Kotányi Aladár	4	Infokomm. rendszerek	HIT	Dr. Tranzisztor Béla
AD1212	Kalapács Attila	3	Vill. energetika	VET	Dr. Apertúra János
DHJ7M4	Lopovszky Károly	3	Szám. rendszerek	AAIT	Dr. Reláció György
96JGTG	Keltai János	4	Szám. rendszerek	AAIT	Dr. Reláció György
85JUHG	Baldai Baldwin	3	Szám. rendszerek	AAIT	Dr. Reláció György

- Érezzük, hogy redundánsan szerepelnek adatok!
 - > **Módosítási anomália:** ha megváltozik egy attribútum érték, több helyen kell módosítani. Ha egy helyen elmarad a módosítás, inkonzisztens lesz (pl. Szakirányfelelős változtatása AAIT esetén)
 - > **Törlési anomália:** ha csak Kalapács Attila-t szeretném törölni, elveszik a Vill. energetika szakirány is (a Tanszék és Szakirányfelelőssel együtt)
 - > **Beszúrási anomália:** nem tudom a Szakirányt és adatait (Tanszék, Szakirányfelelős) felvenni, csak ha felveszek legalább egy hallgatót is (pl. Vill. Energetika, VET, Dr. Apertúra János önmagában)

Relációs adatmodell alapfogalmak

- Funktionális függőség

- > Ha R reláció sorai megegyeznek A1, A2, ..., AN attribútumokon, és ez maga után vonja, hogy meg kell egyezniük egy B attribútumon, akkor B funkcionálisan függ A1, A2, ..., AN-től
 - > Egyszerűbben: Attribútum B funkcionálisan függ A1, A2, ..., AN-től ha belőlük egyértelműen következik B értéke.
 - > Jelölés: A1, A2, ..., AN → B

- Példa

- > NeptunKód → Név
 - > Szakirány → Tanszék, Szakirány felelős
 - > Név, Lakcím, Szül → TAJ

Relációs adatmodell alapfogalmak

- Kulcs
 - > Attribútumok egy halmaza, ha (két feltétellel):
 - 1. minden nem kulcs attribútum funkcionálisan függ a kulcstól: egyediség.
 - Vagyis minden sorra egyedi értéket vesznek fel a kulcs attribútumai együttesen.
 - Vagyis értékei egyértelműen meghatározzák a reláció egy sorát.
 - 2. Ha bármely attribútumot, vagy attribútumokat elhagyjuk, akkor már az első feltétel nem teljesül: minimalitás.
 - Példa:
 - > Hallgató(TAJ, NeptunKód, Név, Lakcím, Szül)
 - Erre igaz:
 - $TAJ \rightarrow NeptunKód, Név, Lakcím, Szül$
 - $NeptunKód \rightarrow TAJ, Név, Lakcím, Szül$
 - $Név, Lakcím, Szül \rightarrow TAJ, Neptun$
 - Vagyis kulcsok a (TAJ), (NeptunKód), (Név, Lakcím, Szül), mert minden igaz, hogy egyértelműen meghatározzák a relációt (egyediek a reláción belül)

Relációs adatmodell alapfogalmak

- Elsődleges kulcs (primary key, PK)
 - > A kulcsok közül kiválasztunk egyet (a többi kulcsjelölt lesz).
 - > Jelölése aláhúzással: Diák(NeptunKód, TAJ, Név, Lakcím, Szül)
- Külső/idegen kulcs (foreign key, FK)
 - > Hivatkozás más tábla elsődleges kulcsára
 - > Példa:
 - Szakirány(SzakirányNév, Tanszék, Felelős)
 - Hallgató(NeptunKód, Név, **SzakirányNév**, Évfolyam)
 - A Hallgató tábla SzakirányNév oszlopa idegen kulcs a Szakirány táblára.
 - > A táblák közötti kapcsolatokat külső kulcsokkal írjuk le
 - > Az adatkezelők ellenőrzik is az idegen kulcs tartalmát.
 - Pl. a Hallgató tábla Szakiránynév oszlopába csak olyan érték kerülhet, ami szerepel a Szakirány tábla SzakirányNév oszlopában
 - Többek között ez biztosítja az adatok integritását!

Relációs adatmodell alapfogalmak

- NULL attribútumérték
 - Amennyiben engedélyezzük, egy attribútum NULL értéket is felvehet.
- Elsődleges kulcs jellemzők
 - > A gyakorlatban fel szoktunk venni egy „mesterséges” mezőt elsődleges kulcsnak
 - > Pl.
 - Alkalmazott (AlkalmazottID, Név, Cím, ...)
 - Árucikk (ÁrucikkID, Név, Ár, Leírás)
 - > Általában
 - AutoIncrement - vagyis az adatkezelő automatikusan kitölti egy egyesével növekvő számértékkel.
 - Vagy GUID (Globaly Unique Identifier) – egy 128 bites „véletlenszerűen” generált érték: olyan nagy a tartomány, hogy gyakorlatilag nulla az ütközés esélye.

Relációs adatmodell alapfogalmak

- Normalizáció

- > Láttuk, hogy bizonyos táblafelépítésnél anomáliák lépnek fel
- > Ennek elkerülésére a relációt (táblát) dekomponálni kell több (relációra) táblára, valamelyen **normál formára**
- > A gyakorlatban a BCNF, illetve a 3NF normál forma használt
 - Erről már előző félévben volt szó
- > A korábbi hallgató-szakirány példa normalizálva
 - Szakirány(SzakirányNév, Tanszék, Felelős)
 - Hallgató(NeptunKód, Név, Évfolyam, Szakiránynév)

NeptunKód	Név	Évfolyam	Szakirány
SD3ER5	Kotányi Aladár	4	Infokomm. rendszerek
AD1212	Kalapács Attila	3	Vill. energetika
DHJ7M4	Lopovszky Károly	3	Szám. rendszerek
96JGTG	Keltai János	4	Szám. rendszerek
85JUHG	Baldai Baldwin	3	Szám. rendszerek

Hallgató tábla

A Hallgató táblában
a SzakirányNév
idegen kulcs

Szakirány tábla

Szakirány	Tanszék	Szakir. felelős
Infokomm. rendszerek	HIT	Dr. Tranzisztor Béla
Vill. energetika	VET	Dr. Apertúra János
Szám. rendszerek	AAIT	Dr. Reláció György

Relációs adatmodell alapfogalmak

- Denormalizáció
 - > Bizonyos esetekben (ritkán) a normalizált struktúrát „elrontjuk”
 - Pl. számított érték tárolása
 - Példa: a szakirányhoz tartozó hallgatók számának tárolása a Szakirány táblában (ne kelljen mindenkor megszámolni a Hallgató táblában)
 - > Csak ha tényleg indokolt!
 - Az előző példában a számított értéket nehéz karbantartani!
- Tranzakció
 - > Műveletek egységbe zárása
 - > ACID elvek betartásához
 - Atomicitás, Konzisztencia, Izoláció, Tartósság
 - (Részletesebben szoftverfejlesztés specializáció: Adatvezérelt rendszerek c. téma)

Objektumrelációs leképezés

Objektumrelációs leképezés

- Az adatbázistervezésnek két módszere van
 - > A) „Hagyományos”: a relációkat dekompozícióval normalizáljuk. Eddig ezt láttuk.
 - > B) Adott egy fogalmi modell (pl. UML osztálydiagram vagy egyed-kapcsolat diagram) →
Az osztályokat/entitásokat és a kapcsolataikat képezzük le táblákra.
- Példa B)-re
 - > Adott az alábbi fogalmi modell (UML osztálydiagram)



- > Ismétlés: első előadáson láttuk, hogy a kapcsolatok „programkódban” hogyan képződnek le (tipikusan legalábbis):
 - **Egy-több kapcsolat (Tárgyfelelős-Tárgy)**: A tárgyfelelős osztályban egy mutató (referencia) lista Tárgy objektumokra, és a Tárgy osztályban egy mutató (referencia) egy Tárgyfelelős objektumra.
 - **Több-több kapcsolat (Hallgató-Tárgy)**: Mindkét osztályban egy mutató (referencia) lista a másik objektumaira.

Objektumrelációs leképezés

- Kérdés – hogyan képezzük le a fogalmi modellt relációs adatbázis sémára
- Folytassuk az előző példát



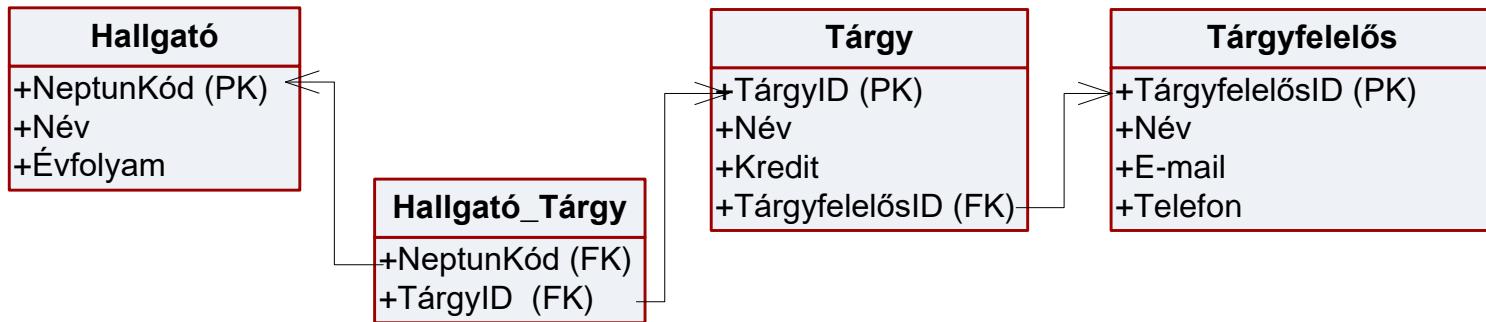
- > Osztály, tagváltozó:
Minden osztályhoz felveszünk egy táblát, a tagváltozóknak egy-egy oszlopot.
- > Egy-több kapcsolat (Tárgyfelelős-Tárgy):
A többes oldal táblájában (Tárgyf) felveszünk egy idegen kulcsot az egyes oldal táblájának (Tárgyfelelős) elsődleges kulcsára.
- > Több-Több kapcsolat (Hallgató-Tárgy):
Felveszünk egy új kapcsolótáblát (Hallgató_Tárgy), amely tartalmaz egy-egy idegen kulcsot az összekapcsolt táblák elsődleges kulcsára.

Objektumrelációs leképezés

- Folytassuk az előző példát



- Leképezve adatbázis táblákra



> Négy táblánk lett: a Hallgató_Tárgy egy kapcsolótábla

Objektumrelációs leképezés

- Folytatás: szemléltetés mintaadatokkal

Hallgató		
NeptunKód	Név	Évfolyam
SD3ER5	Kotányi Aladár	4
AD1212	Kalapács Attila	3
DHJ7M4	Lopovszky Károly	3
96JGTG	Keltai János	4

Tárgy			
TárgyID	Név	Kredit	TárgyfelelősID
1	Valszám	2	2
2	Szoftech	4	1
3	Adatbázisok	3	1

Hallgató_Tárgy	
NeptunKód	TárgyID
SD3ER5	2
SD3ER5	3
AD1212	1
...	...

Kotányi Aladár a Szoftech és az Adatbázisok tárgyakat hallgatja.

Dr. Reláció György a Szoftech és az Adatbázisok tárgyak felelőse. anomáliák?

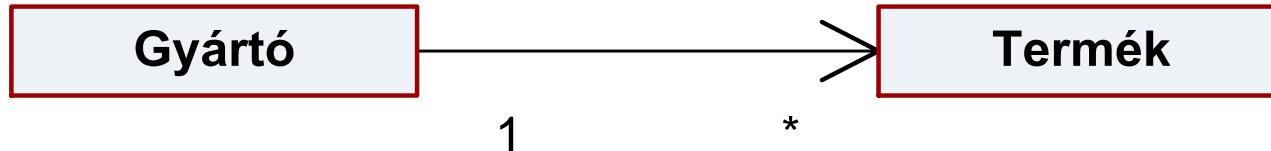
Tárgyfelelős	
TárgyfelelősID	Név
1	Dr. Reláció György
2	Kalapács Attila
3	Lopovszky Károly
4	Keltai János

Ellenőrzük: vannak-e

Az SQL nyelv alapjai

SQL bevezető

- SQL - Structured Query Language: Strukturált lekérdező nyelv.
- Nyelvi elemek két csoportja
 - > DDL - Data Definition Language, adatdefiníciós elemek
 - Pl. Táblák létrehozása, szerkezetük módosítása
 - > DML – Data Manipulation Language, adatkezelési elemek
 - Pl. rekordok beszúrása, módosítása, törlése és lekérdezése
- Példa
 - > Gyártó és termék: Egy gyártó több terméket, egy terméket egy gyártó
 - > Gyártó(GyartID, Nev, Cím)
 - > Termék(TermekID, Nev, Ar, *GyartID*)



SQL - adatdefiníció

- Táblák létrehozása

```
CREATE TABLE Gyarto
(
    GyartoID int NOT NULL PRIMARY KEY,
    Nev nvarchar(50) NOT NULL,
    Cim nvarchar(200) NOT NULL
)

CREATE TABLE Termek
(
    TermekID int NOT NULL PRIMARY KEY,
    Nev nvarchar(50) NOT NULL,
    Ar int NOT NULL,
    GyartoID int NULL REFERENCES Gyarto(GyartoID)
)
```

- CREATE TABLE <táblanév>, majd oszlopok felsorolása. minden oszlopra:
 - > Név, típus, felvehet-e NULL értéket, opcionális megkötés,
 - > az opcionális megkötés lehet: PRIMARY KEY (az adott oszlop az elsődleges kulcs), REFERENCES <hivatkozott tábla neve><hivatkozott oszlop neve>. A REFERENCES-zel idegen kulcsot definiálunk.

SQL - adatmanipuláció

- Sor beszúrása táblába
 - > INSERT INTO táblanév (oszloplista) VALUES (értéklista)

```
INSERT INTO Gyarto (GyartoID, Nev, Cím) VALUES (1, 'Gyártó1', 'Cím1')
INSERT INTO Termek (TermekID, Nev, Ar, GyartoID) VALUES (1, 'Termék1', 100, 1)
```

- Sor(ok) törlése
 - > DELETE FROM táblanév (oszloplista) WHERE <logikai feltétel>

```
-- Összes termék törlése
DELETE FROM Termek
-- Összes 400 forintos termék törlése
DELETE FROM Termek WHERE Ar = 400
```

- Sor(ok) módosítása
 - > UPDATE táblanév oszlopnév1=érték1, oszlopnév2=érték2 WHERE <logikai feltétel>

```
-- 2-es azonosítójú termék nevének és árának módosítása
UPDATE Termek SET Nev = 'Termék2 új', Ar = 220 WHERE TermekID = 2
-- 300 Ft-nál olcsóbb termékek árának növelése
UPDATE Termek SET Ar = Ar*1.2 WHERE Ar < 300
```

SQL lekérdezések

- Lekérdezés
 - > SELECT oszloplista FROM táblalista WHERE <logikai feltétel>
- Logikai feltétel
 - =, <, >, >=, <=, \diamond
 - kif1 BETWEEN kif2 AND kif3
 - kif1 LIKE <sztringminta>, ahol a sztringmintában a következő joker karakterek lehetnek
 - _: egy tetszőleges karakter
 - %: tetszőleges hosszúságú karaktersorozat
 - Példák:
 - „bob”-bal kezdődő szöveg: LIKE ‘bob%’
 - „bob”-ot tartalmazó szöveg: LIKE ‘%bob%’
 - kif IS NULL
 - kif IS NOT NULL
 - AND, OR, NOT
 - EXISTS, NOT EXISTS: a lekérdezés tartalmazza a rekordot

SQL lekérdezések - példák

```
-- A Termek tábla minden  
-- oszlopának listázása ( minden sorra )  
SELECT *  
FROM Termek
```



	TermekID	Nev	Ar	GyartoID
1	1	Termék1	100	1
2	2	Termék2	200	1
3	3	Termék3	400	2
4	4	Termék4	400	NULL

```
-- A Termek tábla TermekID és Nev oszlopának listázása ( minden sorra )  
SELECT Nev, Ar  
FROM Termek
```

```
SELECT *  
FROM Termek  
WHERE Ar > 400
```

```
SELECT *  
FROM Termek  
WHERE Ar > 400 OR Ar < 200
```

```
SELECT *  
FROM Termek  
WHERE Ar BETWEEN 200 AND 300
```

```
SELECT *  
FROM Termek  
WHERE GyartoID IS NULL
```

```
SELECT *  
FROM Termek  
WHERE Nev LIKE '%mék1%'
```

SQL lekérdezések

- Példa1: Listázzuk a gyártókat, meg az általuk gyártott árucikkek neveit!

```
SELECT Gyarto.*, Termek.Nev AS TermekNev  
FROM Gyarto JOIN Termek ON Gyarto.GyartoID = Termek.GyartoID
```

	GyartoID	Nev	Cím	TermekNev
1	1	Gyártó1	Cím1	Temék1
2	1	Gyártó1	Cím1	Temék2
3	2	Gyártó2	Cím2	Temék3

- JOIN, vagy INNER JOIN: két táblát illeszt az ON kulcsszó utáni feltételnek megfelelően
 - A példában a Gyártó táblát a Termék táblával: a Gyártó tábla minden sorához kikeresi a Termék tábla azon sorait, melyre igaz, hogy Gyarto.GyartOID = Termek.GyartOID és az eredményhalmazban ezen sorok minden kombinációja külön sorként megjelenik.

SQL lekérdezések

- Példa2: Listázzuk a termékeket, de a gyártó nevét is jelenítsük meg!

```
SELECT Termek.*, Gyarto.Nev AS GyartoNev  
FROM Termek JOIN Gyarto ON Termek.GyartoID = Gyarto.GyartoID
```

- A „Termék4” nem jelent meg, mert nincs hozzá gyártó megadva (a Gyartoid oszlop értéke NULL)

	TermekID	Nev	Ar	GyartoID	GyartoNev
1	1	Termék1	100	1	Gyártó1
2	2	Termék2	200	1	Gyártó1
3	3	Termék3	400	2	Gyártó2

- Példa3: Listázzuk a termékeket, de a gyártó nevét is jelenítsük meg, és vegyük bele azon termékeket is, amikhez nincs gyártó rendelve.

```
SELECT Termek.*, Gyarto.Nev AS GyartoNev  
FROM Termek LEFT OUTER JOIN Gyarto ON Termek.GyartoID = Gyarto.GyartoID
```

- A LEFT OUTER JOIN illesztés baloldaláról akkor is beleveszi az eredményhalmazba a sort, ha nincs jobbról illeszkedő sor
- A RIGHT OUTER JOIN illesztés jobboldaláról akkor is beleveszi az eredményhalmazba a sort, ha nincs balról illeszkedő sor

	TermekID	Nev	Ar	GyartoID	GyartoNev
1	1	Termék1	100	1	Gyártó1
2	2	Termék2	200	1	Gyártó1
3	3	Termék3	400	2	Gyártó2
4	4	Termék4	400	NULL	NULL

ADO.NET

ADO.NET

- Mi az ADO.NET?
 - > Lehetővé teszi, hogy .NET környezetben fejlesztett alkalmazások relációs adatbázisokat érjenek el.
 - Pl. Oracle, MSSQL, Access, ...
- Mit nyújt?
 - > SQL parancsok futtatása és az eredményhalmaz elérése.
- Alternatívák (.NET)
 - > Van újabb alternatíva, pl. Entity Framework
 - > Az ADO.NET a „hagyományos” módja az adathozzáférésnek, mi ezt nézzük.
- Szemlélet
 - > A legtöbb nem .NET környezetben is nagyon hasonló objektummodellel lehet elérni az adatbázis.

ADO.NET - objektummodell

- Az adathozzáférés legfontosabb objektumai
 - > Connection:
 - Kapcsolatot reprezentál egy adatkezelő felé.
 - Mielőtt bármilyen lekérdezést tudnánk futtatni, kapcsolódni kell az adatbázishoz.
 - > Command:
 - SQL parancsok (és tárolt eljárások futtatását) teszi lehetővé
 - > DataReader
 - Ha egy SQL parancs futtatásának van eredményhalmaza (pl. SELECT), akkor ez teszi lehetővé az adatkezelő által visszaadott adatok elérését.

ADO.NET - Provider alapú architektúra

- Adatkezelő független interfészek és absztrakt œsosztályok
 - > IConnection, DbConnection
 - > ICommand, DbCommand
 - > IDataReader, DbDataReader
- Adatkezelő (provider) függő implementációs osztályok
 - > A fenti absztrakt osztályokból származnak
 - > Pl.:
 - Oracle: OracleConnection, OracleCommand, OracleDataReader
 - MSSQL: SqlConnection, SqlCommand, SqlDataReader
 - ...

ADO.NET – Kapcsolatalapú adathozzáférés

- Lépések (MS SQL providerre példa)
 - > `SqlConnection` objektum példányosítása a kapcsolat paramétereivel
 - > `SqlCommand` objektum létrehozása
 - SQL parancs paraméterben megadása
 - `SqlConnection` objektum megadása paraméterben (a parancs a futtatáskor ezt a connection objektumot használja a kapcsolódáshoz)
 - > Parancs (`SqlCommand`) futtatása
 - > Ha van visszaadott eredményhalmaz (pl. SELECT), az `SqlDataReader`-rel végigmegyünk a rekordokon és kiolvassuk az oszlopok tartalmát
 - > Lezárjuk a kapcsolatot
- Command műveletek
 - > `ExecuteNonQuery` – olyan SQL parancs futtatása, ami nem tér vissza semmivel (pl. INSERT, UPDATE)
 - > `ExecuteScalar` - olyan SQL parancs futtatása, ami csak egy egész számmal tér vissza
 - > `ExecuteReader` – olyan SQL parancs futtatása, ami egy eredményhalmazzal (egy vagy több rekord tér vissza)

ADO.NET – Kapcsolatalapú adathozzáférés

Lekérdezés példa

```
try
{
    // Kapcsolat objektum létrehozása
    conn = new SqlConnection(
        "Data Source=(local); Initial Catalog=SzoftechDB; Integrated security = true");

    // Az adatbázis parancs létrehozása
    SqlCommand command = new SqlCommand("SELECT TermekID, Nev, Ar FROM Termek", conn);

    // A kapcsolat megnyitása
    conn.Open();

    // Az adatok lekérdezése és kiiratása
    reader = command.ExecuteReader();
    while (reader.Read())
    {
        Console.WriteLine("{0}\t{1}\t{2}", reader["TermekID"].ToString(),
            reader["Nev"].ToString(), reader["Ar"].ToString());
    }
}
finally
{
    if (reader != null) reader.Close();
    if (conn != null) conn.Close(); // minden esetben bontsuk a kapcsolatot !!!
}
```

ADO.NET – Kapcsolatalapú adathozzáférés

- Jellemzők
 - > A DataReader-rel egyszer lehet előre haladva végignavigálni az eredményhalmaz sorain (pl. nem lehet visszalépni).
 - > reader["<olszlopnév>"] az aktuális sor adott oszlopának tartalmát adja vissza
 - > Költséges erőforrás a megnyitott adatbázis kapcsolat!!!!
 - A kapcsolatot mielőbb zárjuk le!
 - Módosító műveleteknél a parancs(ok) futtatása után azonnal
 - Lekérdezéskor azonnal, miután a DataReaderrel felolvastuk az adatokat.
 - Miért költséges?
 - Általában korlátozott az egyidőben megnyitható kapcsolatok száma: ha sok felhasználó kapcsolódik egyidőben a kiszolgáló alkalmazásunkhoz, elfogyhatnak a szabad kapcsolatok
 - Sokszor adatkezelő licencdíjat is a max. engedélyezett párhuzamos kapcsolatok után kell fizetni.

ADO.NET – Kapcsolatalapú adathozzáférés

Beszúrás példa

```
SqlConnection conn = null;
try
{
    // Kapcsolódás az adatbázishoz
    conn = new SqlConnection("Data Source=.\SQLExpress; Initial Catalog = SzoftechDB;
Integrated Security = True");

    conn.Open(); // A kapcsolat megnyitása

    // Paraméterek megadása SQL-INJECTION!!!
    SqlParameter pTermekId = new SqlParameter("@TermekId", termkekId);
    SqlParameter pNev = new SqlParameter("@Nev", nev);
    SqlParameter pAr = new SqlParameter("@Ar", ar);

    // A parancs létrehozása
    SqlCommand command = new SqlCommand(
        "INSERT INTO Termek(TermekId, Nev, Ar) VALUES(@TermekId, @Nev, @Ar)");

    command.Parameters.Add(pTermekId);
    command.Parameters.Add(pNev);
    command.Parameters.Add(pAr);

    // Adatbázis kapcsolat megadása
    command.Connection = conn;
    int res = command.ExecuteNonQuery();

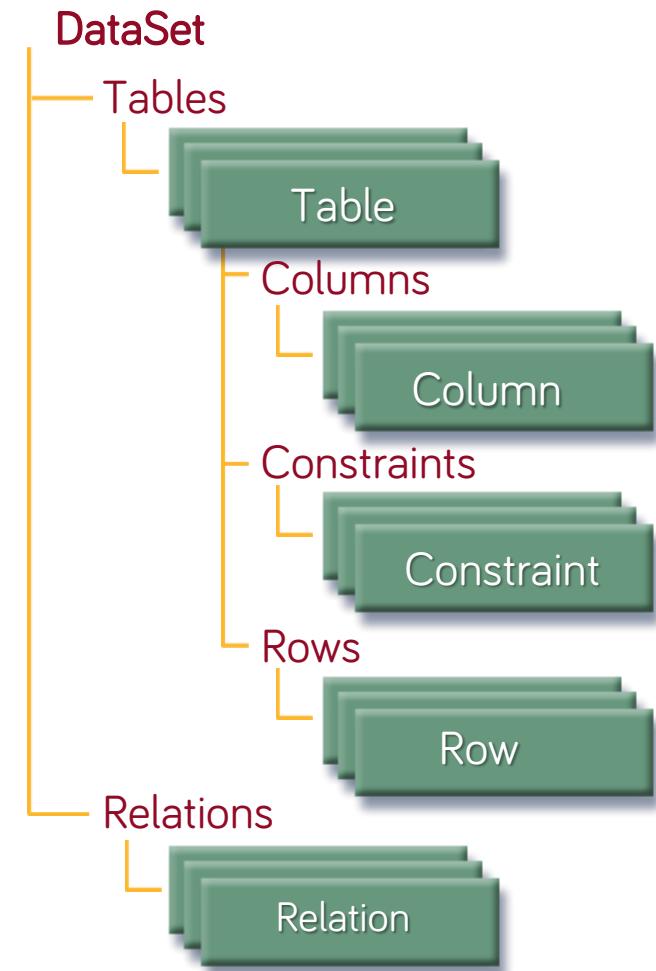
}
finally { if (conn != null) conn.Close(); }
```

ADO.NET – „Kapcsolatnélküli” adathozzáférés

- „Kapcsolatnélküli,” adathozzáférés
 - > Szokás „DataSet alapúnak” is nevezni (ha DataSet-tel dolgozunk)
 - > Tipikusan a következőképpen dolgozunk adatokkal
 - Adatok lekérdezése (SELECT) és felolvasása az adatbázisból, a kapcsolat bontása
 - Adatok megjelenítése a felhasználó számára, pl. táblázatban (grid)
 - A felhasználó szerkeszti az adatokat (ez „sok” idő lehet)
 - A felhasználó elmenti a változtatásokat: ennek során kapcsolódunk az adatbázishoz, és elmentjük a változtatásokat (INSERT, UPDATE, DELETE).
 - > Megoldandó problémák
 - A felolvasást követően a memóriában tárolni kell az adatokat
 - A felhasználó módosításait is tárolni kell, amíg el nem mentjük (mi az új, módosított, törlendő). Ezt csak akkor, ha nem mentünk minden egyes változtatást azonnal vissza az adatbázisba.
 - > Megoldás a problémáakra
 - Vagy leprogramozzuk valami egyedi megoldással
 - Dolgozhatunk DataSet-tel, ami erre lett kitalálva

ADO.NET – DataSet alapú adathozzáférés

- Mi a DataSet
 - > Egy memóriabeli adatbázis (táblákkal, oszlopokkal, sorokkal, táblák közötti kapcsolatokkal (idegen kulcsok))
- Mit nyújt számunkra
 - > A táblában nyilvántartja az adatbázisból feltöltött adatokat
 - > A táblák sorai szerkeszthetők: az adatbázisba való visszamentésig minden változást nyilvántart



ADO.NET – DataSet alapú adathozzáférés

- Lépések

- 1) Lekérdezés

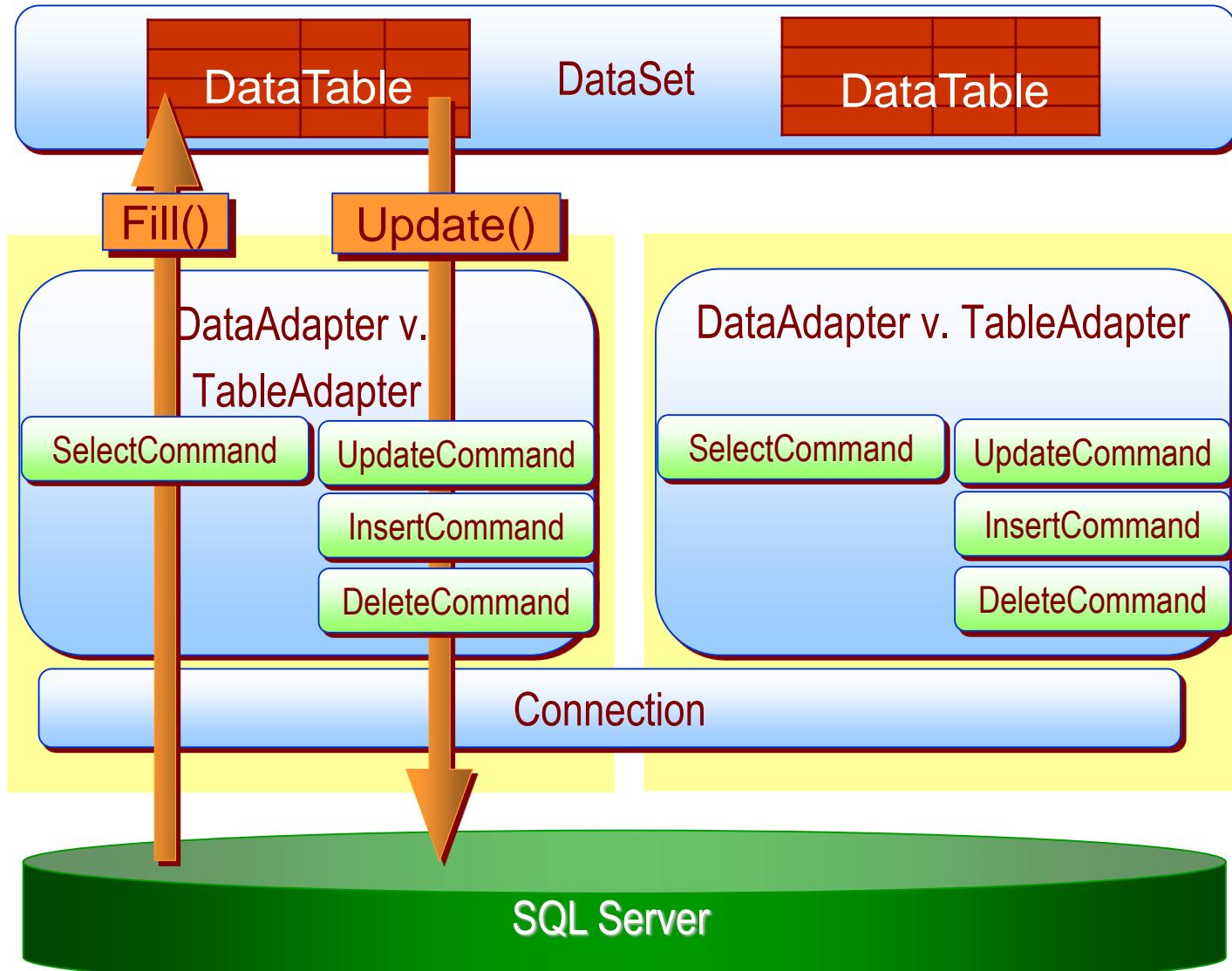
- Kapcsolat megnyitása
- DataSet adott tábláinak feltöltése adatbázisból lekérdezés lapján
- Kapcsolat lezárása

- 2) Az adatok megjelenítése a DataSet táblái alapján, illetve ezek módosítása (a memóriában)

- 3) Változtatások visszamentése adatbázisba

- Kapcsolat megnyitása
- DataSet táblák változtatásainak visszamentése. A mentés a háttérben soronként történik:
 - minden új sorra lefut egy INSERT
 - minden módosult sorra egy UPDATE
 - minden törölt sorra egy DELETE
- Kapcsolat lezárása

ADO.NET – DataSet alapú adathozzáférés



ADO.NET – DataSet alapú adathozzáférés

```
// Kapcsolat objektum létrehozása
SqlConnection conn = new SqlConnection(
    "Data Source=(local); Initial Catalog=SzoftechDB;...");

// DataSet objektum létrehozása
DataSet dsSzoftech = new DataSet();

// Data adapter létrehozása: a dataset feltöltését, és a változások visszamentését
// végzi majd.
SqlDataAdapter adapter = new SqlDataAdapter("SELECT * FROM Termek", conn);
adapter.InsertCommand = new SqlCommand(< egy INSERT parancs >);
adapter.UpdateCommand = new SqlCommand(< egy UPDATE parancs >);
adapter.DeleteCommand = new SqlCommand(< egy DELETE parancs >);

// DataSet "Termek" táblájának feltöltése: lefut az adapter SELECT parancsa
// A kapcsolat automatikusan felépül és bomlik
adapter.Fill(dsSzoftech, "Termek");

// A konzolra írjuk ki a "Termek" tábla 0. sorának "Nev" oszlopának tartalmát
Console.WriteLine(dsSzoftech.Tables["Termek"].Rows[0]["Nev"].ToString());

// A "Termek" tábla 0. sorának "Ar" oszlopa legyen 999
dsSzoftech.Tables["Termek"].Rows[0]["Ar"] = 999;

// A változtatása visszaírása az adatbázisba: a módosított sorra lefut az
// UpdateCommandban megadott SQL parancs. A kapcsolat felépül majd bomlik is.
adapter.Update(dsSzoftech, "Termek");
```

Kitekintés

- Probléma
 - > A DataSet még mindig nyers SQL utasításokon dolgozik
 - > Mi lenne ha a lekérdezéseket is C#-ban írhatnánk, C# osztályokon dolgozva
- Egy lehetséges megoldás: Entity Framework
 - > A legelterjedtebb magas szintű ORM keretrendszer .NET-hez (Microsoft)
 - > Ennek a tárgynak nem anyaga
 - Bővebben a szoftverfejlesztés specializáción
 - Adatvezérelt rendszerek c. tárgy

Ellenőrző kérdések

- Ismertesse a relációs adatmodell alapjait (reláció, tábla, attribútum, sor, relációs séma)!
- Egy példán keresztül mutassa be a relációs modellekben fellépő anomáliákat!
- Példákkal illusztrálva ismertesse a funkcionális függőség, kulcs, elsődleges kulcs és az idegen kulcs fogalmát!
- Mutasson példát relációs adatmodell normalizációra! Mi a célja a normalizációnak?
- Mi a célja az ORM (objektumrelációs) leképezésnek? Rövid magyarázattal mutasson egy példát a leképezésre!
- Mi az SQL nyelv célja?
- Mutasson példát SQL nyelv alapú adatdefinícióra!
- Mutasson példákat SQL nyelv alapú adatmanipulációra (beszúrás, módosítás, törlés, lekérdezés)!

Ellenőrző kérdések

- Mi az ADO.NET, és milyen három fontosabb osztályt definiál? Mi ezen osztályok feladata?
- Mik a kapcsolatalapú adathozzáférés lépései?
- Adjon meg C# példakódot kapcsolatalapú adathozzáférésre (rövid magyarázattal)!
- Mire kell figyelni az adatbázis-kapcsolatra vonatkozóan? Miért?
- Ismertesse a „kapcsoltnélküli” adathozzáférés lépésein, megoldandó problémáit általánosságában!
- Ismertesse a DataSet fogalmát!
- Ismertesse a DataSet alapú „kapcsoltnélküli” adathozzáférés lépésein!
- Adjon meg C# példakódot DataSet alapú „kapcsoltnélküli” adathozzáférésre (rövid magyarázattal)!

Szoftvertechnikák

Architekturális tervezés



Automatizálási és
Alkalmazott
Informatikai Tanszék

Tartalom

Tervezési szemlélet

Alapvető architekturális
minták

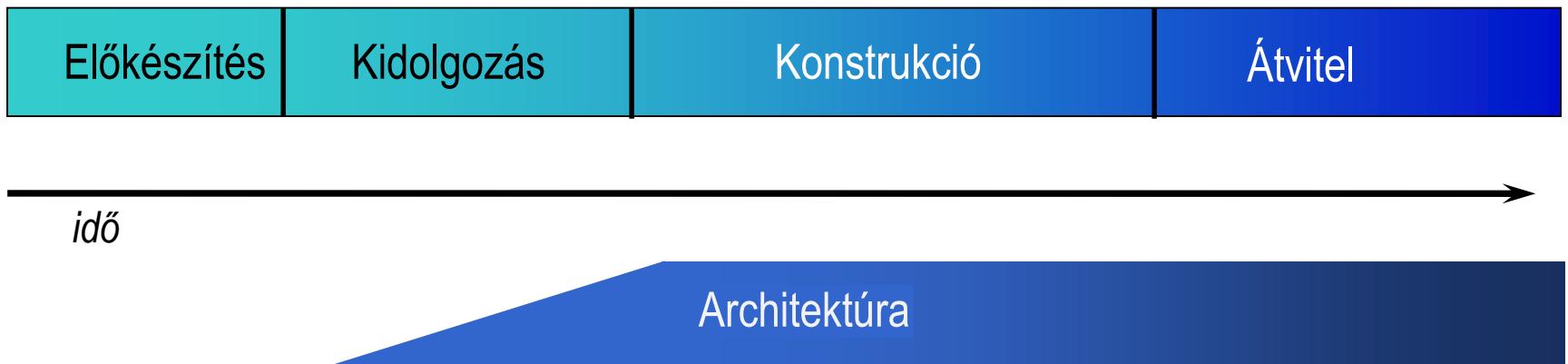
- > Rétegelés
- > Vállalati Információs
Rendszerek architektúrái
- > Document View
- > MVC
- > Pipes and filters

Mi az architektúra?

- A szoftverrendszer szervezése, strukturálása
- A strukturális elemek kompozíciója és együttműködése

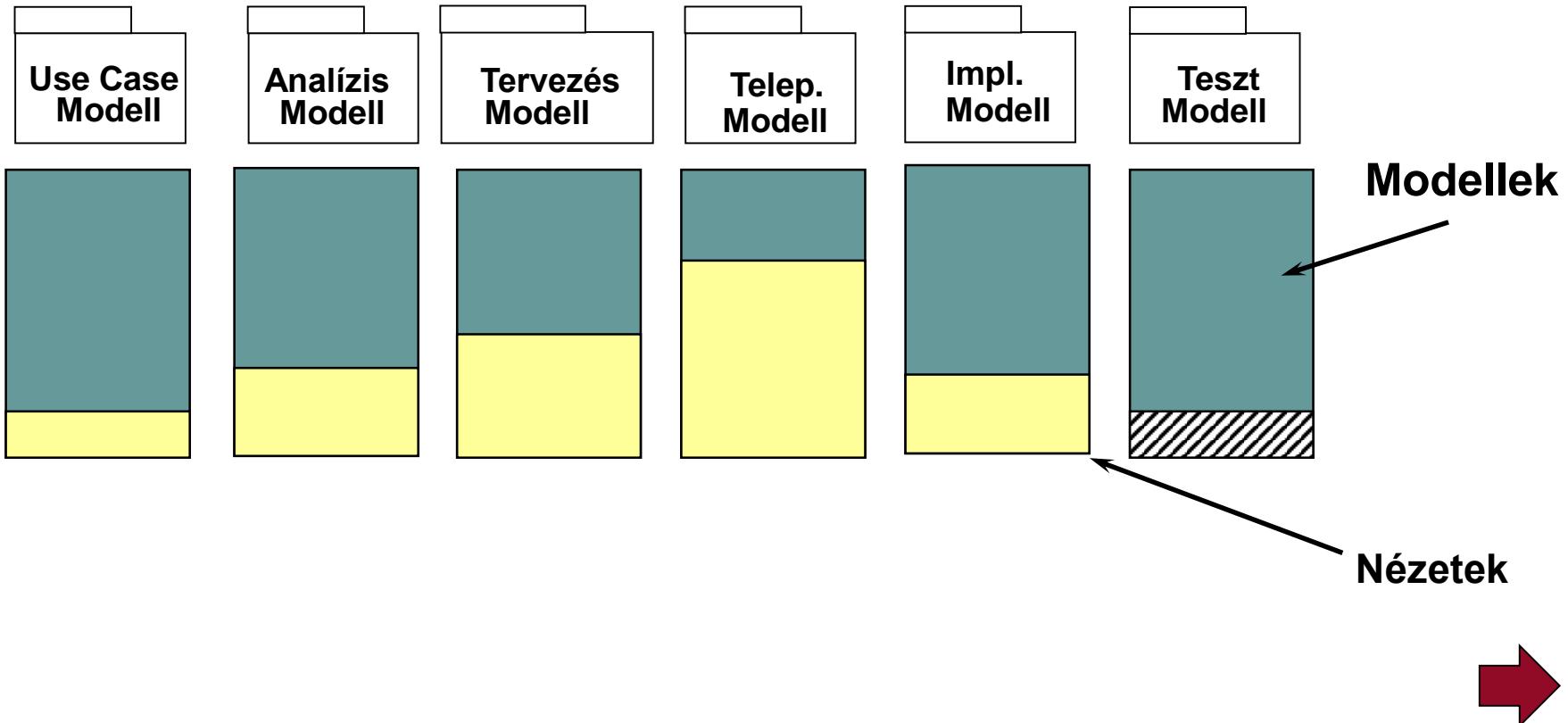
Architektúra-centrikusság

- A modellek láthatóvá teszik, specifikálják, megalkotják és dokumentálják az architektúrát.
- A Unified Process a futtatható architektúra sorozatos finomítását írja elő



Az architektúra és a modellek

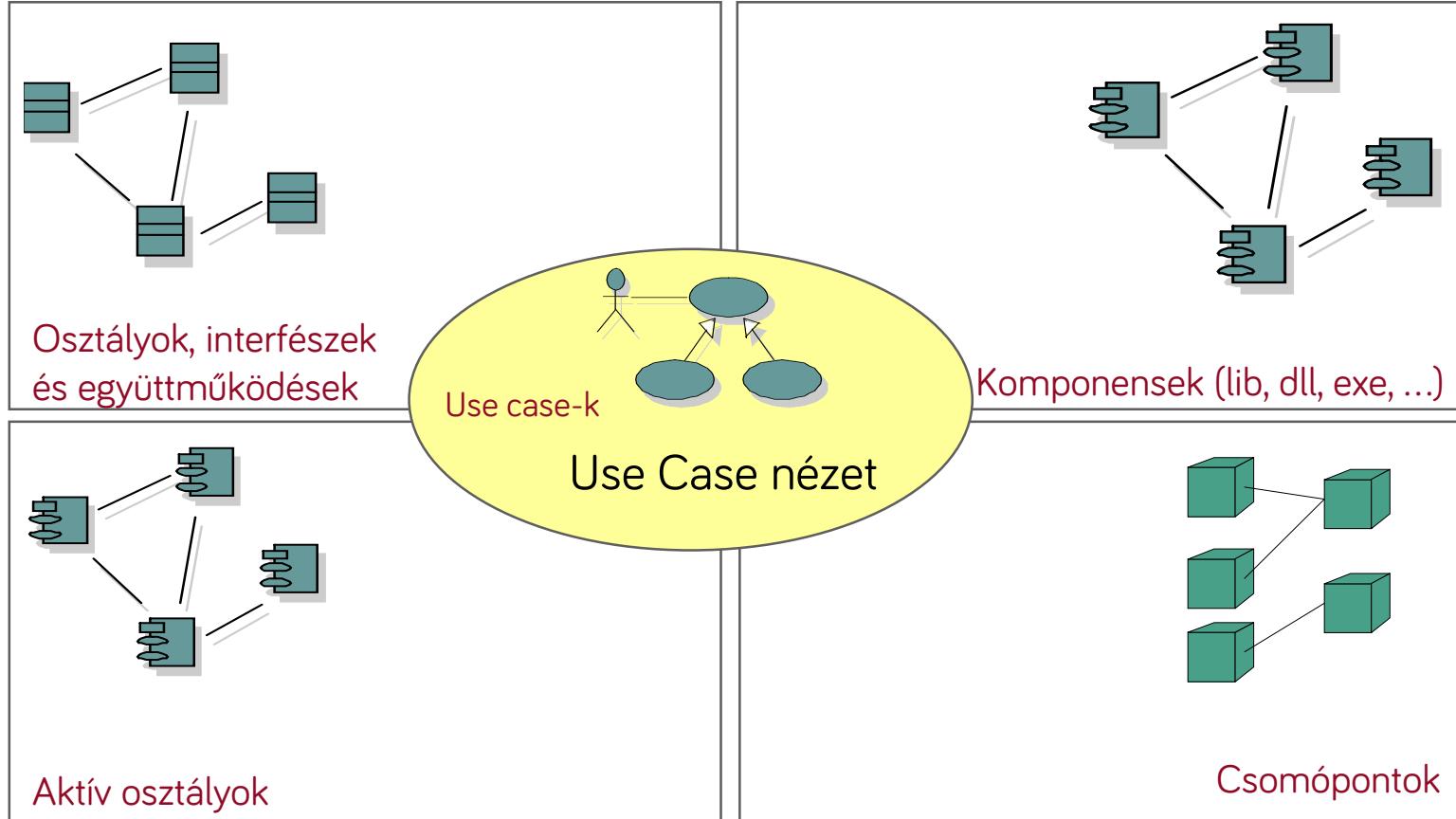
- A modell nézetek architekturális információtartalma



A modell nézetek architekturális információtartalma

- Nem minden terv/modell architektúra!
- Az architekturális elem meghatározó a rendszer felépítése, teljesítménye, stb. szempontjából. Amiből már nem lehet elvenni ahhoz, hogy megértsük és elmagyarázzuk a rendszer működését.
- „From mud to structure.”

Az architektúra nézetei



Elrendezés
Csomag, alrendszer

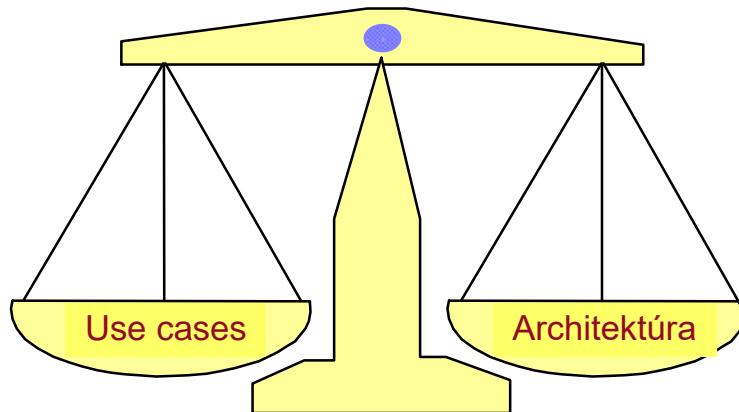
Dinamika
Kölcsönhatás,
Állapotok

Mi az architektúra?

- **Használati eset**
 - > Funkcionális követelmények.
- **Tervezési vagy logikai**
 - > Főbb csomagok (névterek), alrendszerek, osztályok.
- **Implementációs nézet**
 - > Komponensek, dll-ek, exe, forráskódok szervezése.
- **Processz nézet**
 - > Konkurrens aspektus, folymatok, szálak, deadlock, startup, shutdown, teljesítmény, skálázhatóság.
- **Telepítési nézet**
 - > A futtatható és egyéb komponensek mely számítási csomópontokra kerülnek.

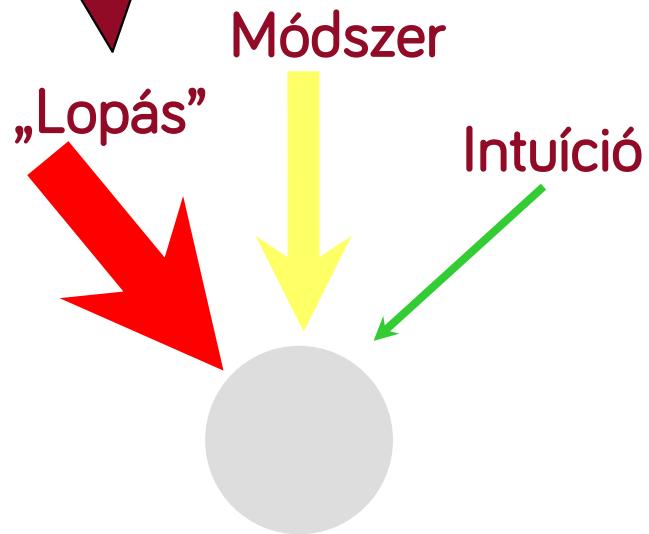
Az architektúra és a funkcionalitás

- Az architektúrának és a funkcionalitásnak egyensúlyban kell lenni!
 - > Funkció: működjön jól! \longleftrightarrow Architektúra: legyen jól strukturált!

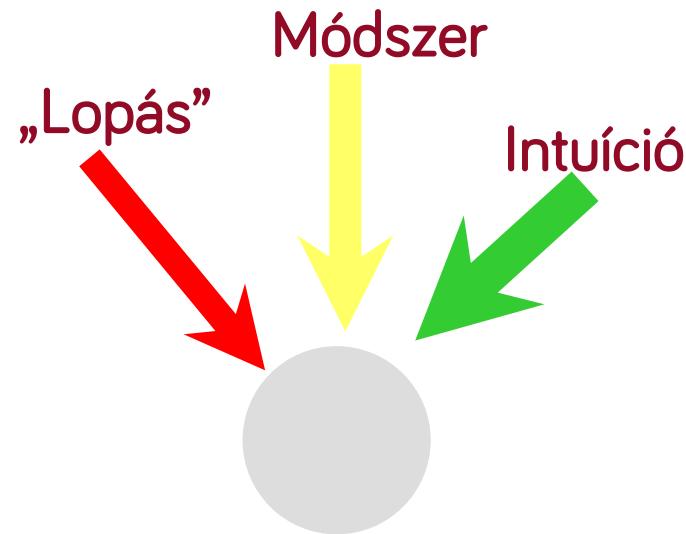


Az architektúra forrásai

Ezzel foglalkozunk
most!



Klasszikus rendszer



Példátlanul ritka rendszer

„Lopás”: átvesszük a már jól bevált megoldásokat.

Architekturális minták: jól bevált architekturális megoldások.

Rétegek (layers)

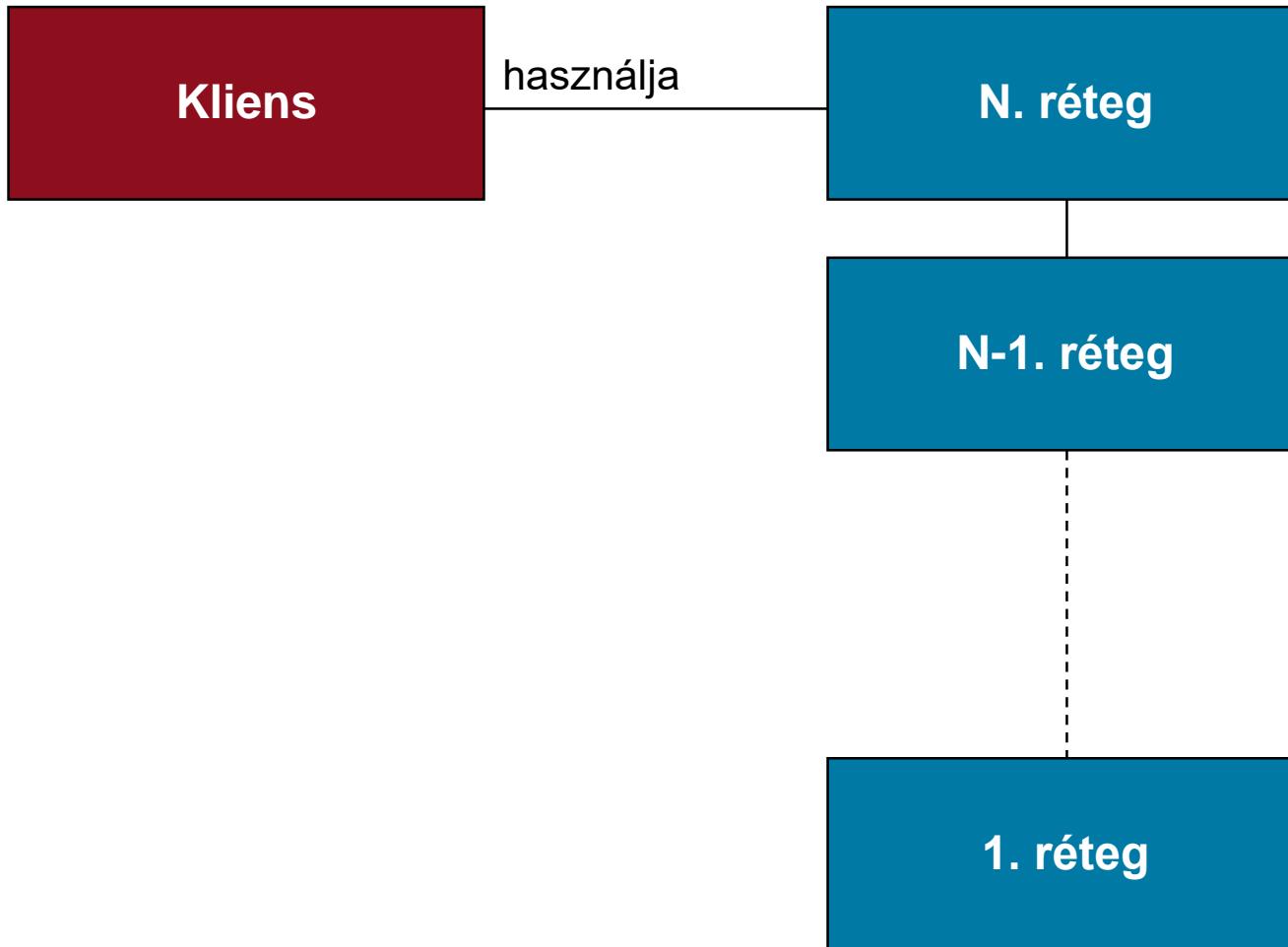
Rétegelés

- Legalapvetőbb szervezési elv
 - > A kódot rétegekbe szervezzük
- Hol fordul elő
 - > A batch világban nem volt
 - > OS felépítése: driver - OS (API) – alkalmazás
 - > Hálózati protokollok
 - > Vállalati információs rendszerek
 - > Stb.

Példa : OSI vs. TCP/IP

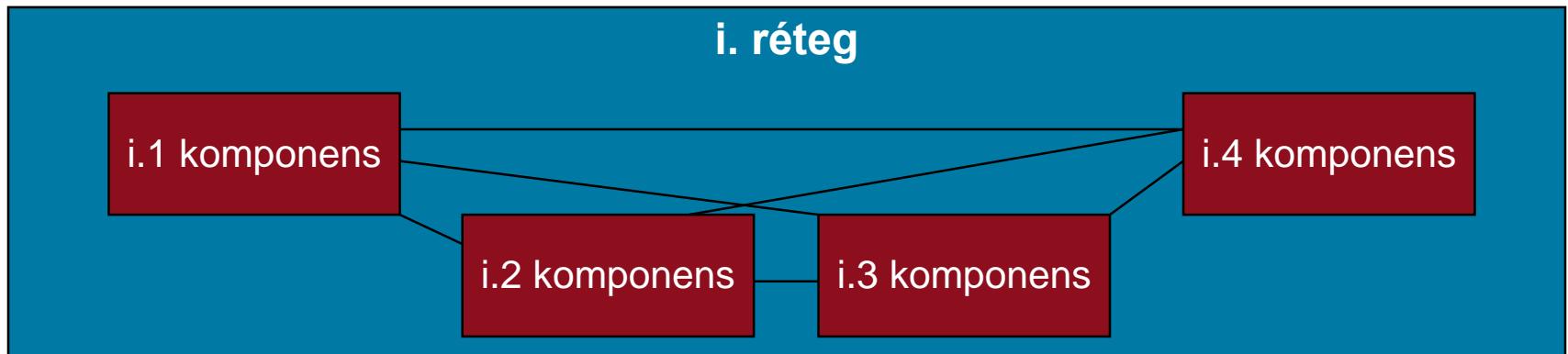


Szigorú rétegelés



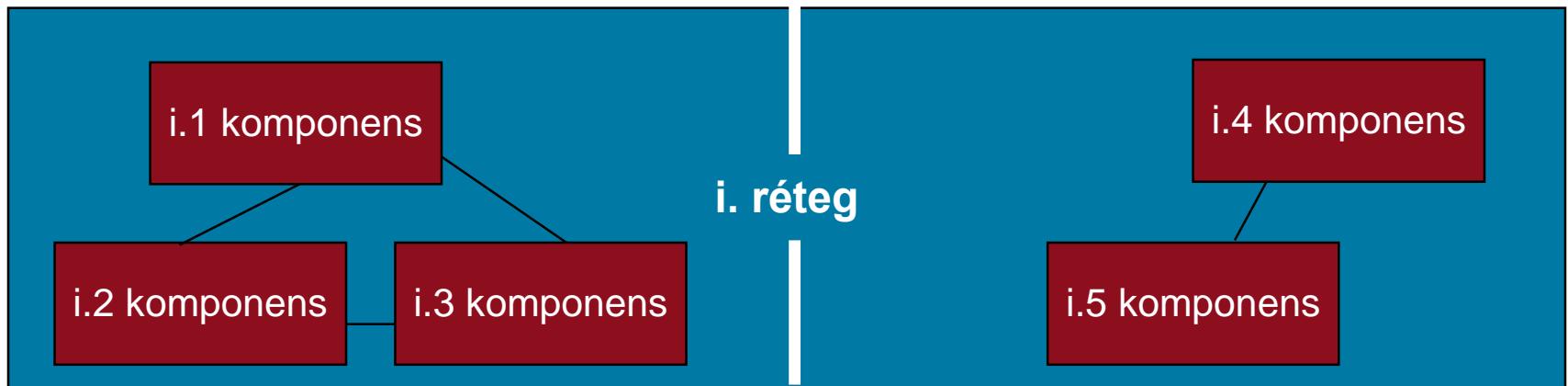
Szigorú rétegelés

- Az i. réteg
 - > szolgáltatásokat nyújt az i+1. rétegnek
 - > A saját szolgáltatásait az i-1. réteg szolgáltatásaira építve valósítja meg
- Az egyes rétegeken belül azonban tetszőleges függőségi viszonyok:



Függőleges partícionálás

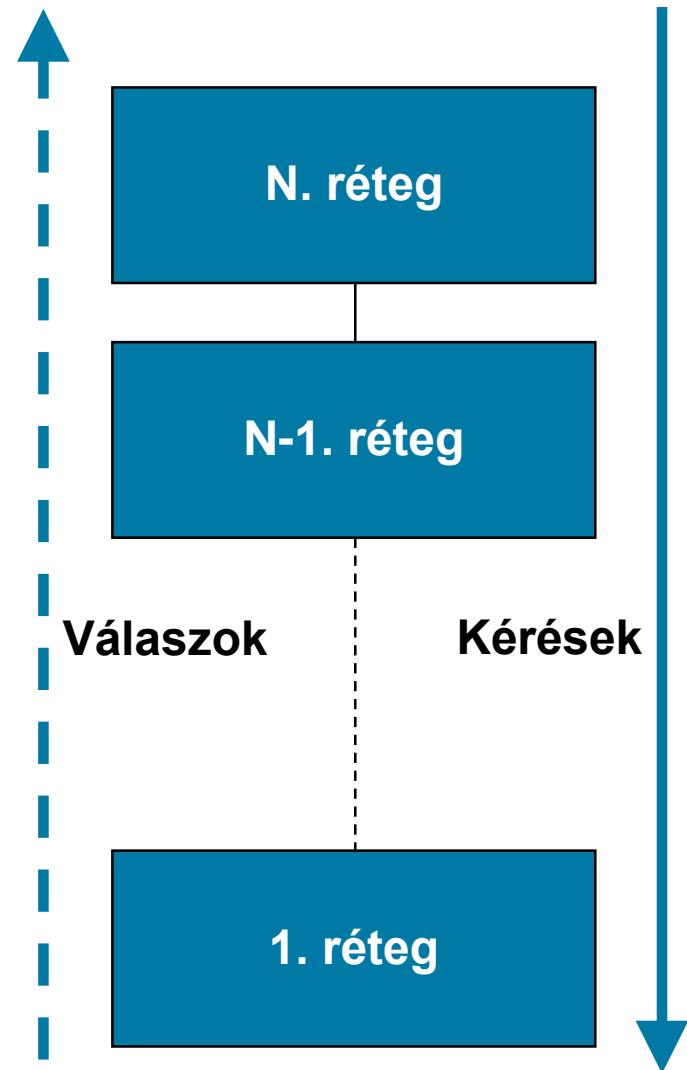
- Az egymástól független komponenseket egy rétegen belül csoportosíthatjuk:



- Előnye: a részek egymástól függetlenül cserélhetők, fejleszthetők.

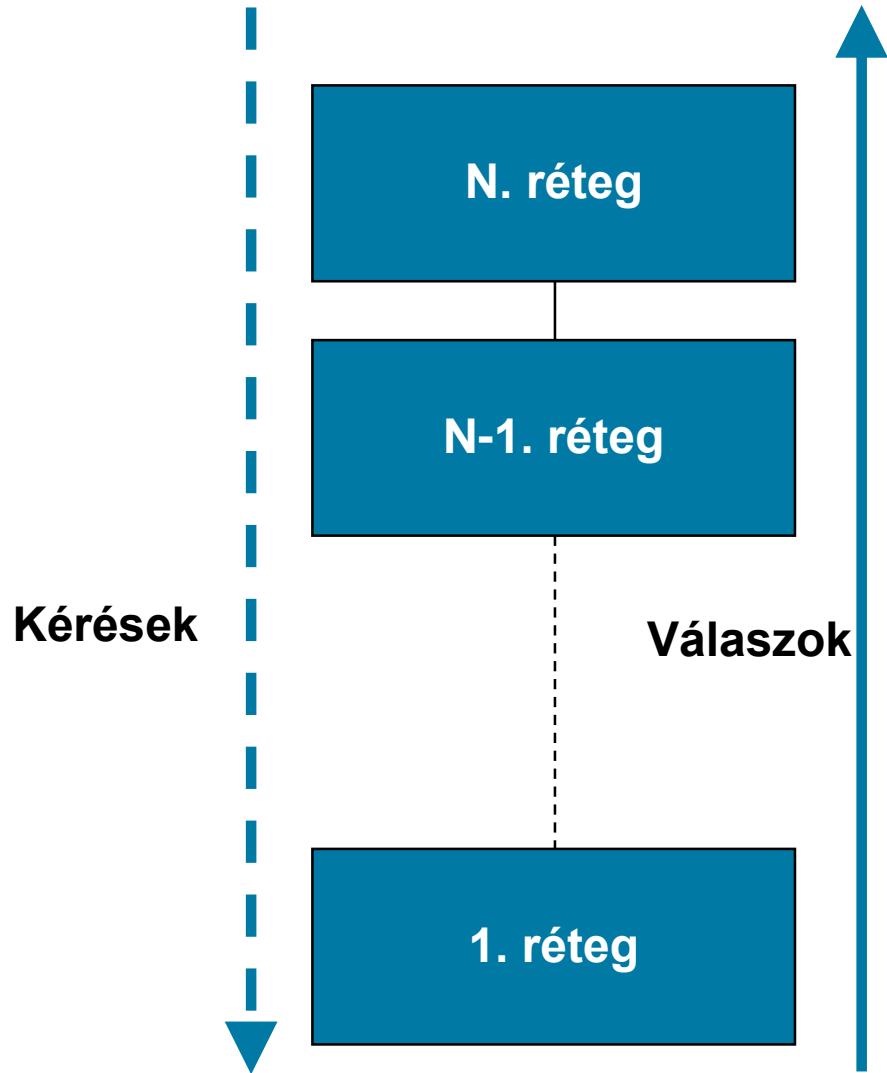
1. Forgatókönyv

- Kezdeményezés:
 - > „Fentről le”
 - Driverek
 - „Polling”



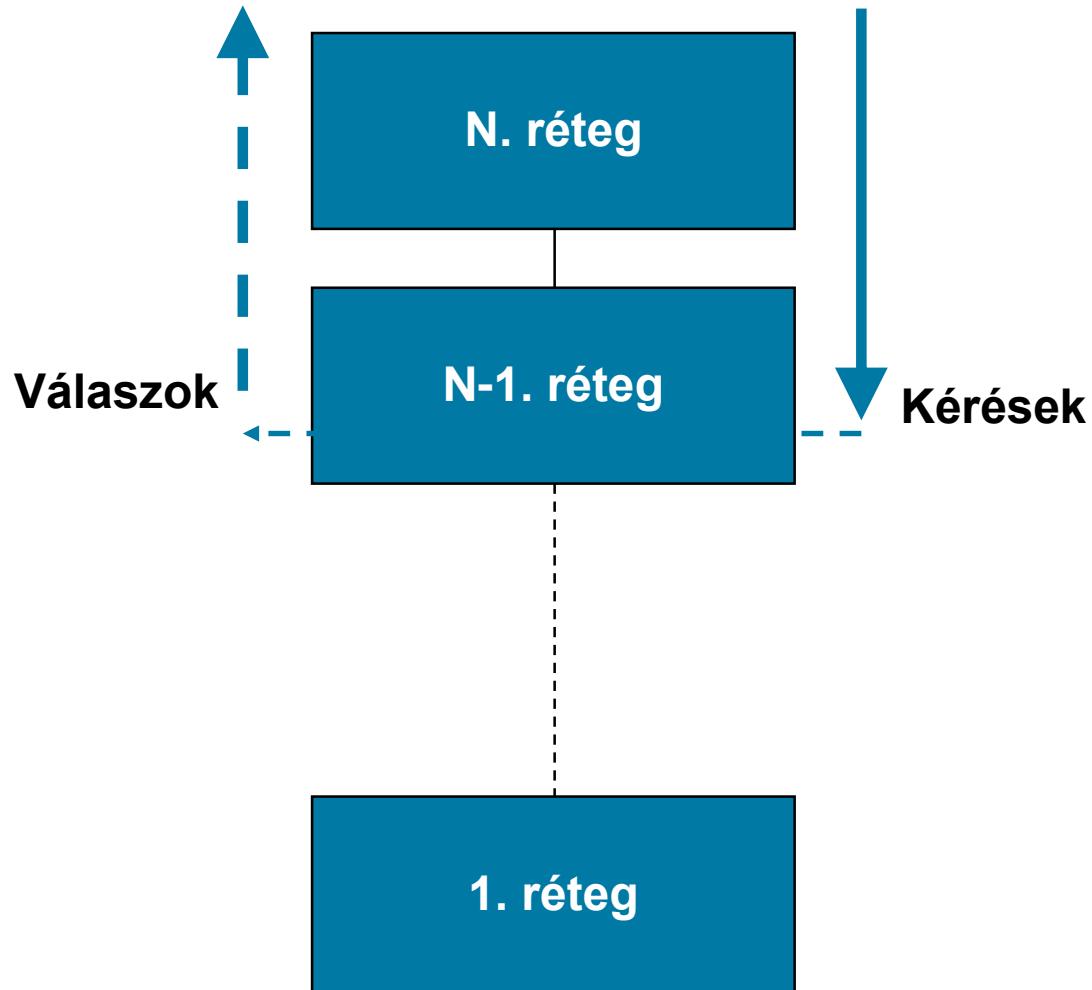
2. Forgatókönyv

- Kezdeményezés:
 - > „Lentről fel”
 - Driverek
 - „Interrupt driven”



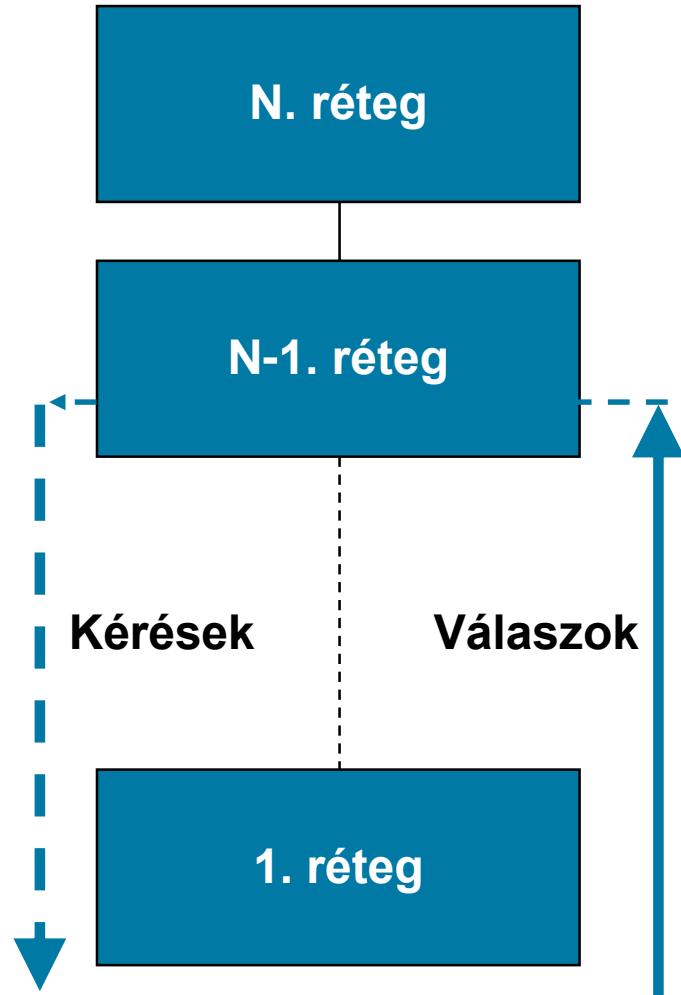
3. Forgatókönyv

- Megszakított 1.
 - > „Cache”
 - > Hiba esetén



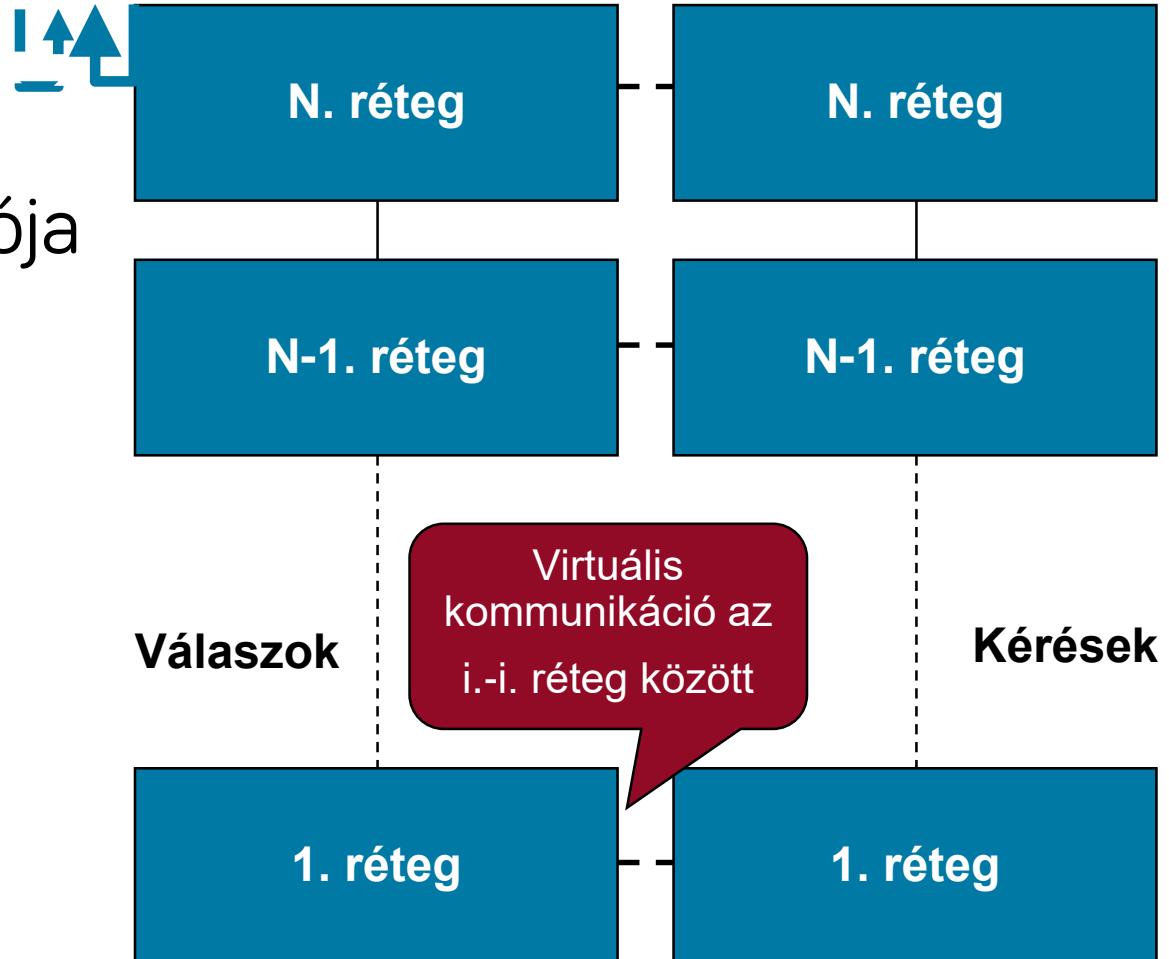
4. Forgatókönyv

- Megszakított 2.
 - > Alsóbb rétegek kommunikációja
 - > Hibakezelés



5. Forgatókönyv

- Két rétegelt architektúra kommunikációja
- Hálózati stack



WCF példa

Rétegelés előnyök

- Túl komplex a feladat: vezessünk be egy új absztrakciós szintet (új rétegbe) és oldjuk meg erre építve
- Egy réteg működése önmagában is megérthető, nem kell a többöt is megértsük
 - > (pl. FTP-hez nem kell az Ethernetet réteg megértése)
- Párhuzamos fejlesztés
 - > Az interfészek kialakítás után a rétegek egymástól függetlenül párhuzamosan fejleszthetők.
 - > Az egyes rétegek eltérő technológiai ismeretei igényelnek
 - (pl. UI fejlesztő, adatszakértő, stb.), nem kell mindenkinél mindenhez értenie a fejlesztőcsapatban.
- Az egyes rétegek automata „unit” tesztelhetősége általában könnyebben megoldható.

Rétegelés előnyök

- Egy rétegre építve sok szolgáltatás építhető (többször is felhasználható):
 - > pl. Socketre építve Webszerver, FTP, stb.
- Egy réteg mögött kicserélhető, bővíthető a többi réteg, így az adott réteg szolgáltatásai új kontextusban is felhasználhatók
 - > (pl. FTP Etherneten megy, de mobil környezetre is kiterjeszthető)

Hátrányok

- Egyszerűbb feladatnál felesleges komplexitás
- A változások sok esetben kaszkádoltan végigvonulnak az összes rétegen, több helyen kell módosítani
 - Pl. felveszünk egy új adatbázis mezőt, akkor a GUI és az adatbázis réteg között minden réteget módosítani kell
- Teljesítmény

Információs rendszerek (vállalati rendszerek) rétegei

Információs rendszerek

- Nincsenek rétegek
 - > Batch rendszerek
- Kétrétegű architektúra
 - > Kliens-szerver
- Hárromrétegű architektúra
- SOA architektúra (nem lesz részletesen)

Információs rendszerek

- Kétrétegű architektúra
 - > Megosztott adatbázis (file-ok, DBMS)
 - > Alkalmazások (hagyományos vagy 4GL nyelven)
 - > Felépítés:

Alkalmazások

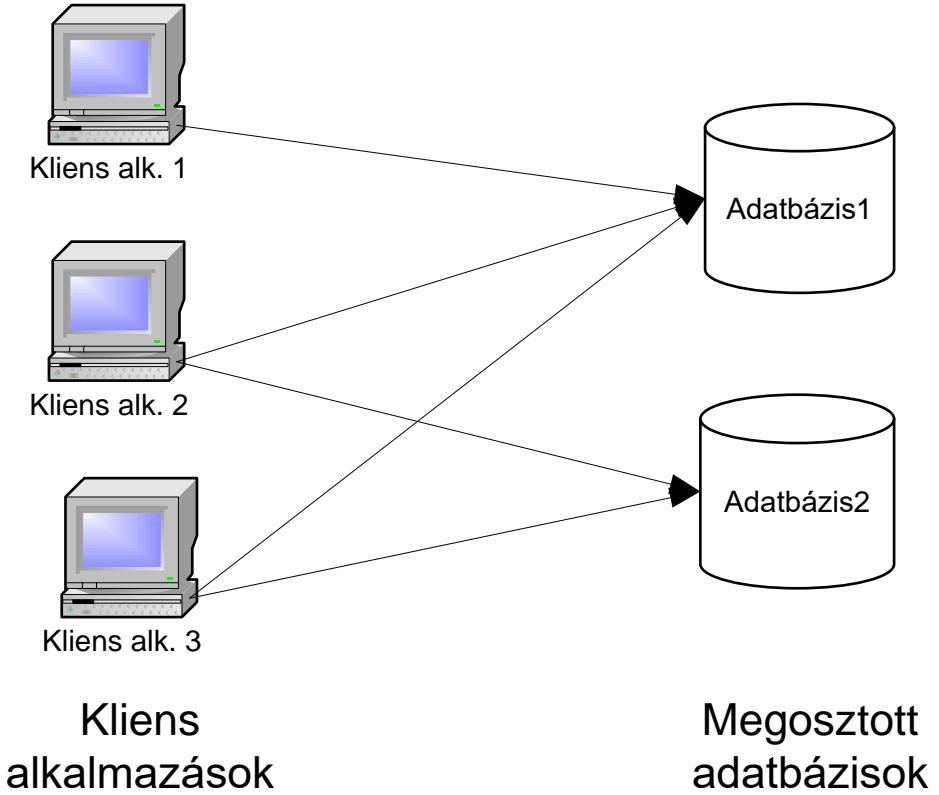
Megosztott adatbázis

- > Elterjedt még a „kliens-szerver” elnevezés is.

Kétrétegű architektúra

- A kliens alkalmazások közvetlenül hozzáférnek az adatbázisokhoz
- Alapgondolata: az adat megosztott, a feldolgozás/megjelenítés elosztott (kliensek).

(BKV útvonal-adatbázis példa)



Kétrétegű architektúra előnyök

- A már meglévő adatokat több szempontból (nézetből) is megjeleníthetjük – több alkalmazást írhatunk
- Az adatkarbantartás elkülöníthető az alkalmazásoktól
- Kitűnő RAD támogatás (.NET, régebben Delphi, VB, PowerBuilder), adatkötés -> rendkívül népszerű.

Kétrétegű architektúra - hátrányok

- Alaprobléma: hova tegyük az üzleti logikát (adatintegritás, üzleti szabályok, számítások, validálás)?
 - > A, Tehetjük a kliens alkalmazásba
 - ⌚ Ha több alkalmazást írunk, akkor duplikálódik az üzleti logika, több helyen kell karbantartani, stb.
 - ⌚ Az adatbázist nem lehet megváltoztatni a régi alkalmazásokkal való kompatibilitás miatt
 - ⌚ Keveredik az üzleti logika a GUI-val
 - > B, Tehetjük az adatbázisba (tárolt eljárásba)
 - ⌚ Nem minden adatkezelő rendszer támogatja
 - ⌚ Ha van is: nem objektum orientált, korlátozott lehetőségek

Kétrétegű architektúra - hátrányok

- Megjelent a webes világ: jó lenne, ha az üzleti logikához csak egy webes megjelenítést lehetne kapcsolni a vastag kliens mellé: ezt nem teszi lehetővé egyszerűen
 - > Mert az üzleti logika és a kliensoldali logika nincs különválasztva.
- Egyéb
 - > Az alkalmazás az adatbázis fizikai felépítését is figyelembe kell vegye, pedig neki csak a szemantikát kellene.
 - > Az adatbázist gyakran denormalizáljuk teljesítmény okokból (optimalizálás): zavaró, a kliensben is megjelenik.

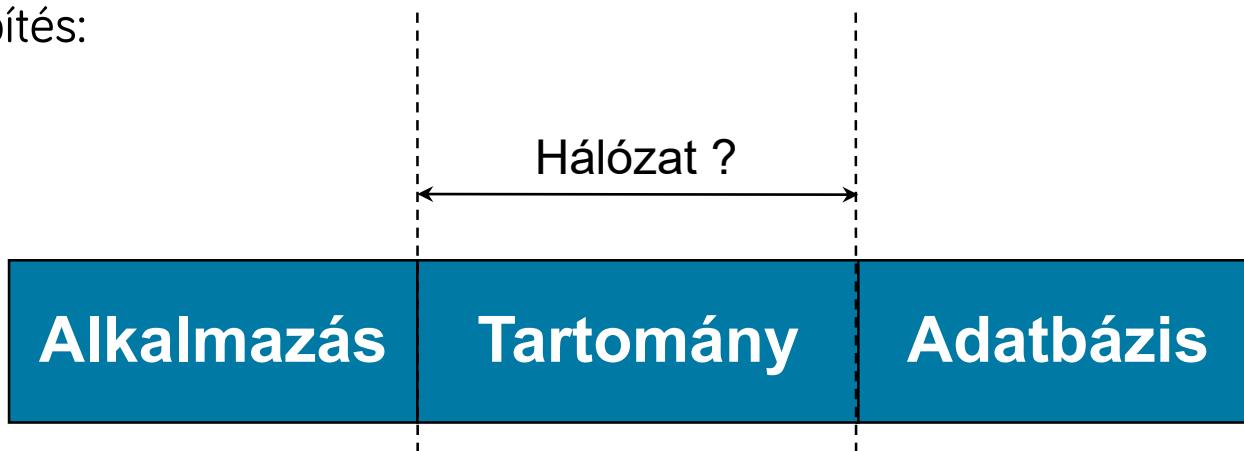
Kétrétegű architektúra

- Mikor használjuk
 - > Egyszerűbb alkalmazások esetében használjuk bátran
 - Ekkor nem éri meg extra rétegekkel vacakolni
 - > Ha nem akarunk üzleti logikát több alkalmazásban, alrendszerben felhasználni (vagy ha igen akkor azt nem tudjuk/akarjuk tárolt eljárásokban implementálni)

Háromrétegű architektúra

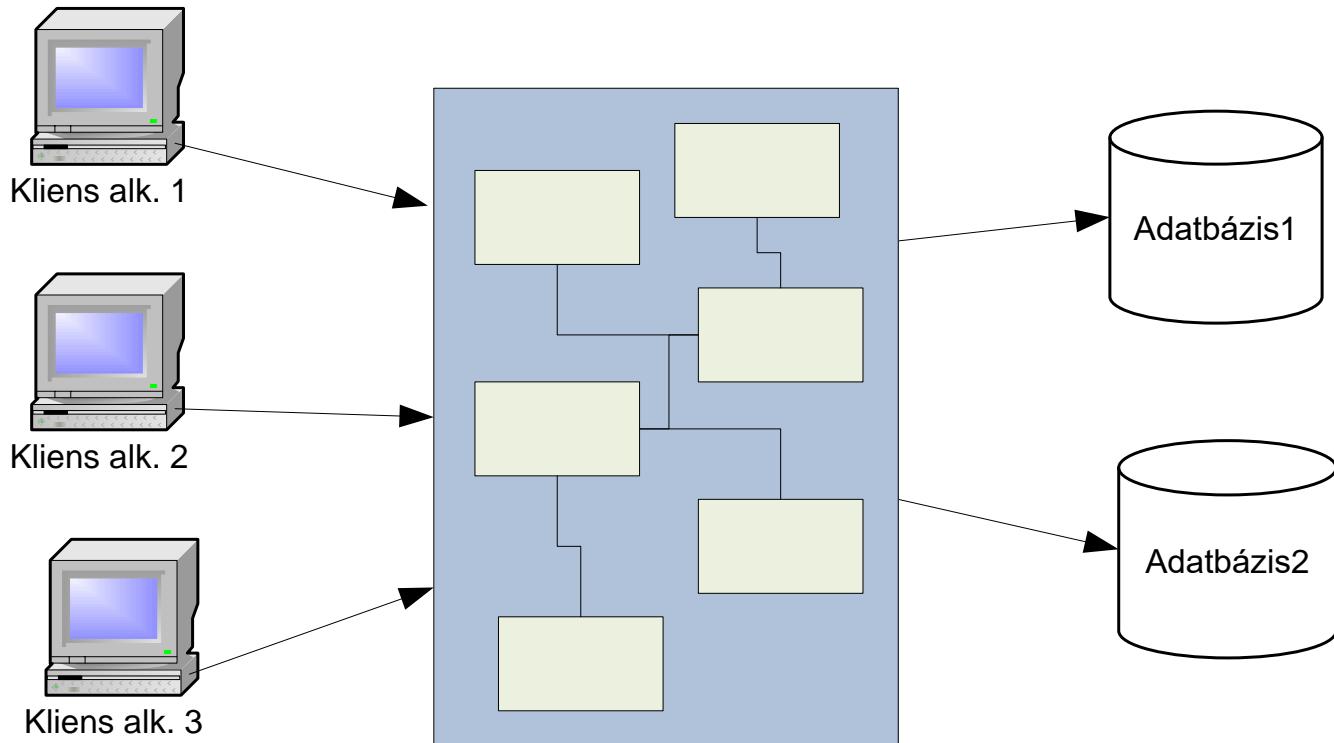
- Háromrétegű architektúra
 - > Alkalmazás (Külső séma)
 - Alkalmazáslogikai alréteg, tipikusan UI (felhasználói felület)
 - > Tartomány (Domain , Koncepcionális séma) – üzleti logika
 - > Adatbázis (Belső séma)

Felépítés:



A kétrétegűhöz képest bevezettünk egy középső réteget, ami az üzleti logikát tartalmazza.

Háromrétegű architektúra



Alkalmazások
(külső séma)

Tartomány
(Fogalmi
séma)

Megosztott
adatbázisok
(belő tárolási
séma)

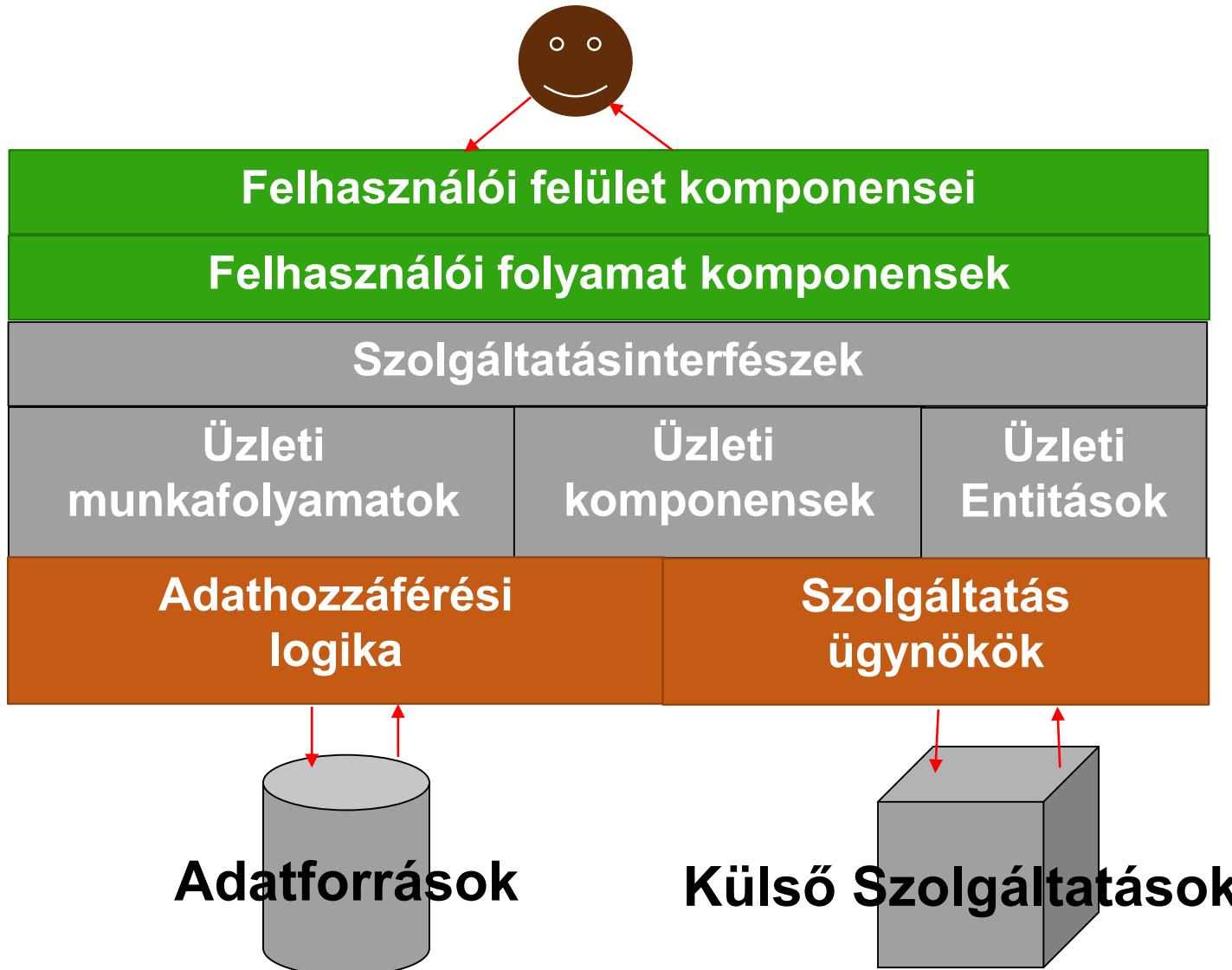
Háromrétegű architektúra előnyök

- Az üzleti logikai részek
 - > Nincsenek duplikálva az alkalmazásokban, mint a kétrétegűnél
 - > Nincsenek szétszórva a kliensben/GUI-ban
 - > Mivel független a GUI/klienstől, könnyebben újrafelhasználható (pl. webes és vastag kliens)
 - > Automatizált tesztek könnyebben készíthetők (pl. NUnit)
- Az alkalmazás kevésbé függ a fizikai adatszerkezetől, az adatbázisok helyétől
 - > Az adatbázis átszervezhető az alkalmazástól függetlenül
 - > A referenciális integritás alkalmazásfüggetlenül kezelhető
- A középső réteg cache-elheti az adatokat*
- Ipari támogatottság*

Háromrétegű architektúra

- Hátrányok
 - > Extra komplexitás, többletmunka
 - > Nagyobb erőforrásigény
 - > Az üzleti logikai komponenseket le kell képezni relációs modellre, ami nehéz
 - Az objektum-orientált adatbázisok jó felhasználási területe, de ezek teljesítőképessége kétséges

A „Microsoft háromrétegű modellje”



A „Microsoft háromrétegű modellje”

- Nem csak „Microsoft”, minden környezetben ezek szoktak megjelenni
- Magyarázat
 - > **Adathozzáférési logika** (Data Access Layer – DAL): Elrejti az adatkezelő rendszer (pl. MSSQL, Oracle) függőségeket a magasabb rétegek elől.
 - > **Szolgáltatásügynökök**: A külső rendszerekkel való kommunikációt valósítják meg.
 - > **Üzleti entitások**: Adatokat reprezentálnak (pl. megrendelés, vásárló), melyek az egyes üzleti logikai komponensek között mozgathatók.
 - > **Üzleti komponensek**: Az üzleti logikát valósítják meg (pl. megrendelés felvétele, listázás, üzleti számítások, stb.)

A „Microsoft háromrétegű modellje”

- Üzleti munkafolyamatok
 - > Bonyolultabb üzleti folyamatok esetén a lépéseket definiálják, és a folyamat levezénylése a feladata
- Szolgáltatásinterfészek
 - > Egy vékony réteg: a kliensek csak ezen keresztül férhetnek hozzá az üzleti szolgáltatásokhoz, így az üzleti komponensek és a munkafolyamatok szabadon átszervezhetők (nem kell a klienseket módosítani).

A „Microsoft háromrétegű modellje”

- Felhasználói felület komponensei
 - > A felhasználói felület űrlapjai, ablakai a rajtuk levő vezérlőelemekkel és GUI logikával.
- Felhasználói folyamat komponensek
 - > Feladata a bonyolultabb felhasználói folyamatok, interakciók levezénylése (mely ablakok, webes űrlapok jelenjenek meg egymás után, stb.).
 - > Kisebb jelentőségű, ritkán használt.

„Rétegek” és „rétegek”

- Layers and tiers, avagy logikai és fizikai rétegek
- Layer
 - > Logikai réteg
 - > Ezt tervezük először, erről volt eddig szó
 - > Célszerű lehet külön modulba (pl. .NET szerelvény vagy Java package) kitenni
- Tier
 - > Fizikai réteg más gépen fut, hálózati kommunikáció
 - > A logikai régekre mondjuk meg, mely gépeken fussenak
- Ajánlások
 - > Ne keverjük a kettőt, gyakran a „profik” is összekeverik
 - > Webes világban adott, előny a központi telepítés és karbantartás

DOCUMENT-VIEW ARCHITEKTÚRA

Bevezető

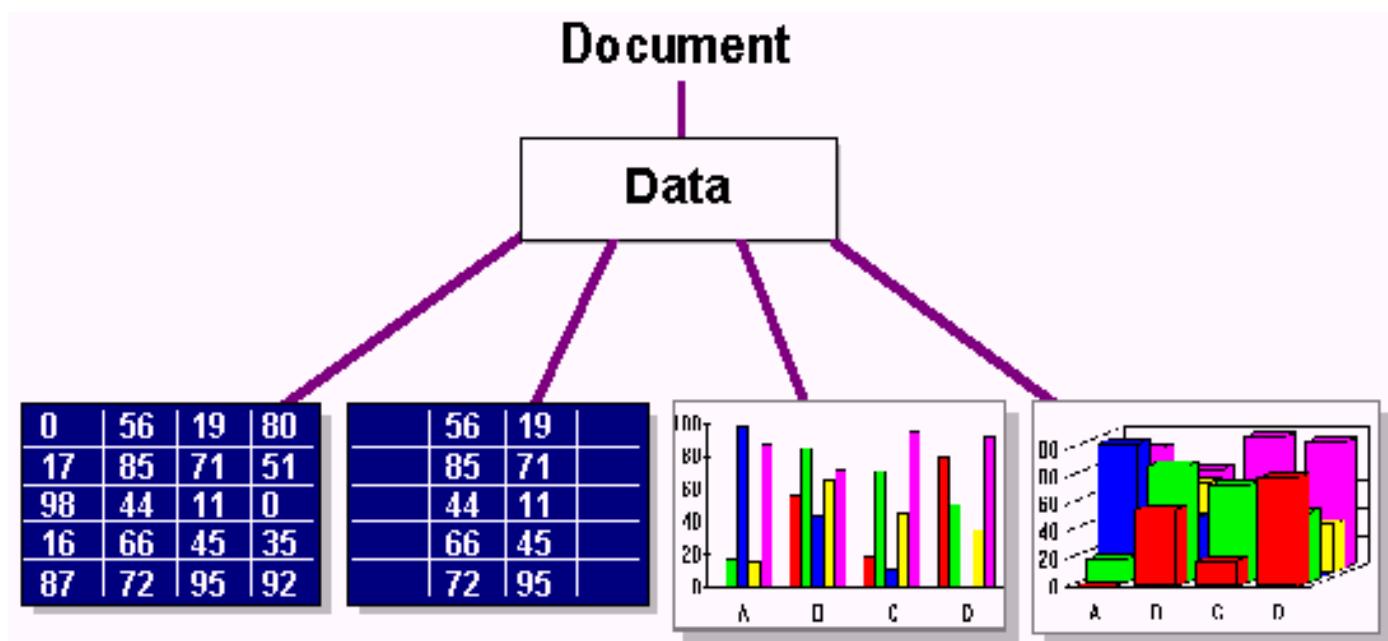
- Az alkalmazások többsége adatokat jelenít meg, melyet a felhasználó módosíthat.
- Alapigazság: ne keverjük bele a GUI-ba az alkalmazáslogikát
 - Rettentő hosszú, áttekinthetetlen, nehezen karbantartható form/windows forráskód
 - Az alkalmazáslogika nem újrafelhasználható
 - Az alkalmazáslogikára nem lehet automatizált teszteket írni
- Válasszuk külön az adatok **kezeléséért** felelős kódot az adatok **megjelenítéséért** felelős kódtól.
- Egy lehetséges megoldás a Document-view architektúra
 - > Alternatíva pl. a Model-View-Controller (MVC)

A Document-view architektúra szereplői

- Document (dokumentum)
 - > Feladata az **adatok tárolása, menedzselése.**
 - > Modellnek is szokás nevezni.
 - > Olyan osztály(ok), melyek az adatokat tagváltozóikban tárolják, és olyan tagfüggvényekkel rendelkeznek, melyek kezelik ezeket az adatokat (pl. Load, Save), és elérhetővé teszik más osztályok számára (pl. a View részére)
- View (nézet)
 - > Feladata az adatok **megjelenítése** a dokumentum adatai alapján és a **felhasználói interakciók kezelése** (pl. menük, egér, billentyűzet).
 - > A felhasználói interakciók során általában a nézet a dokumentum tartalmát módosítja
 - > A view általában egy ablakként, vagy tabfölként jelenik meg a kliensalkalmazásokban

Támogatja a következőket

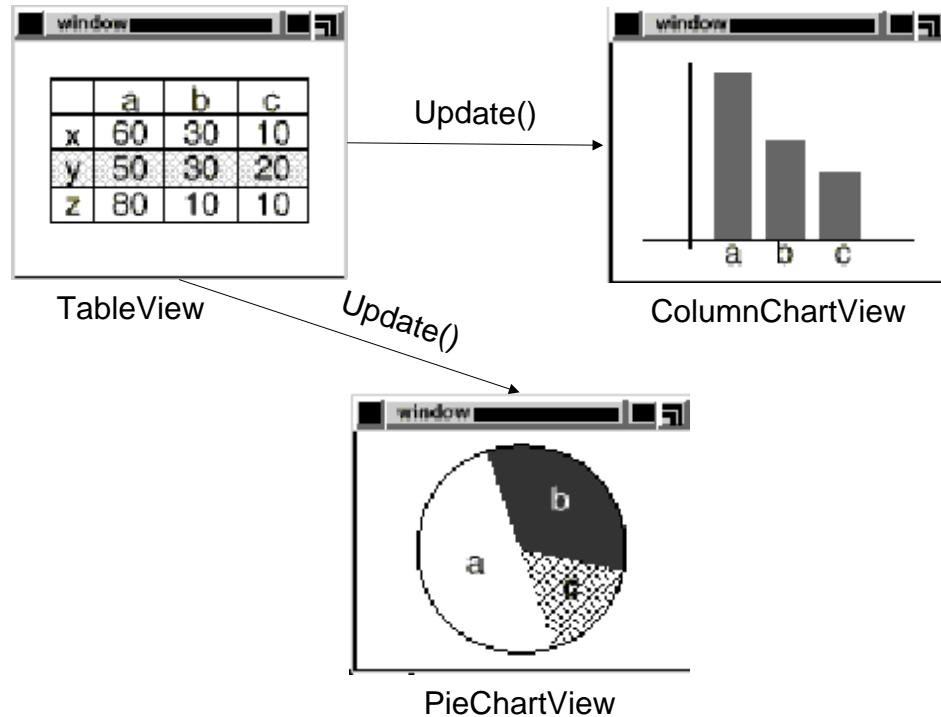
- Több dokumentum egyidejű megnyitása
 - > pl. Firefox tabok, MS Word dokumentumok
- Egy dokumentumhoz több és többféle nézet kapcsolódhat
 - > pl. Excel, de általában a View/New Window menüvel



Possible Views

Több nézet

- Egy dokumentumnak több nézete
 - > Meg kell oldani, hogy az egyes view-k konzisztens nézetét jelenítsék meg az adatoknak. Például ha a felhasználó megváltoztatja az egyik nézeten az adatokat, frissíteni kell a többit. Hogyan?
 - Közvetlen függvényhívással (minden nézet tartalmaz egy referenciát az összes többire)?



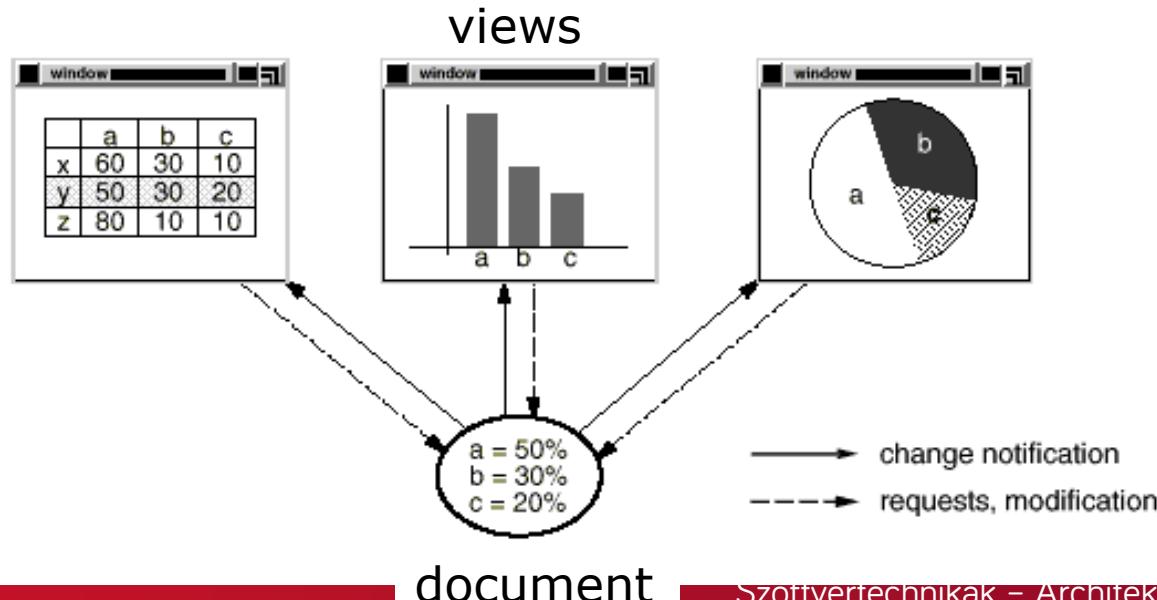
Nézetek frissítése

- Közvetlen függvényhívás hátrányok
 - > Függőség a konkrét osztálytól.
 - Pl. a `TableView` függ a `ColumnChartView` és a `PieChartView` osztályuktól
 - > Ha új nézetet szeretnék bevezetni, minden nézet osztályt módosítani kell
 - > A modell (üzleti logika) nem újrafelhasználható, mert össze van vonva a megjelenítéssel.
 - Cél lenne, hogy úgy jelenjen meg, hogy ne legyen benne hivatkozás egy (konkrét) megjelenítési osztályra sem, mert akkor fel tudnánk több helyen használni
 - > Nehéz karbantartani, továbbfejleszteni, újrafelhasználni, mert túl szoros a csatolás az osztályok között

A nézetek frissítése

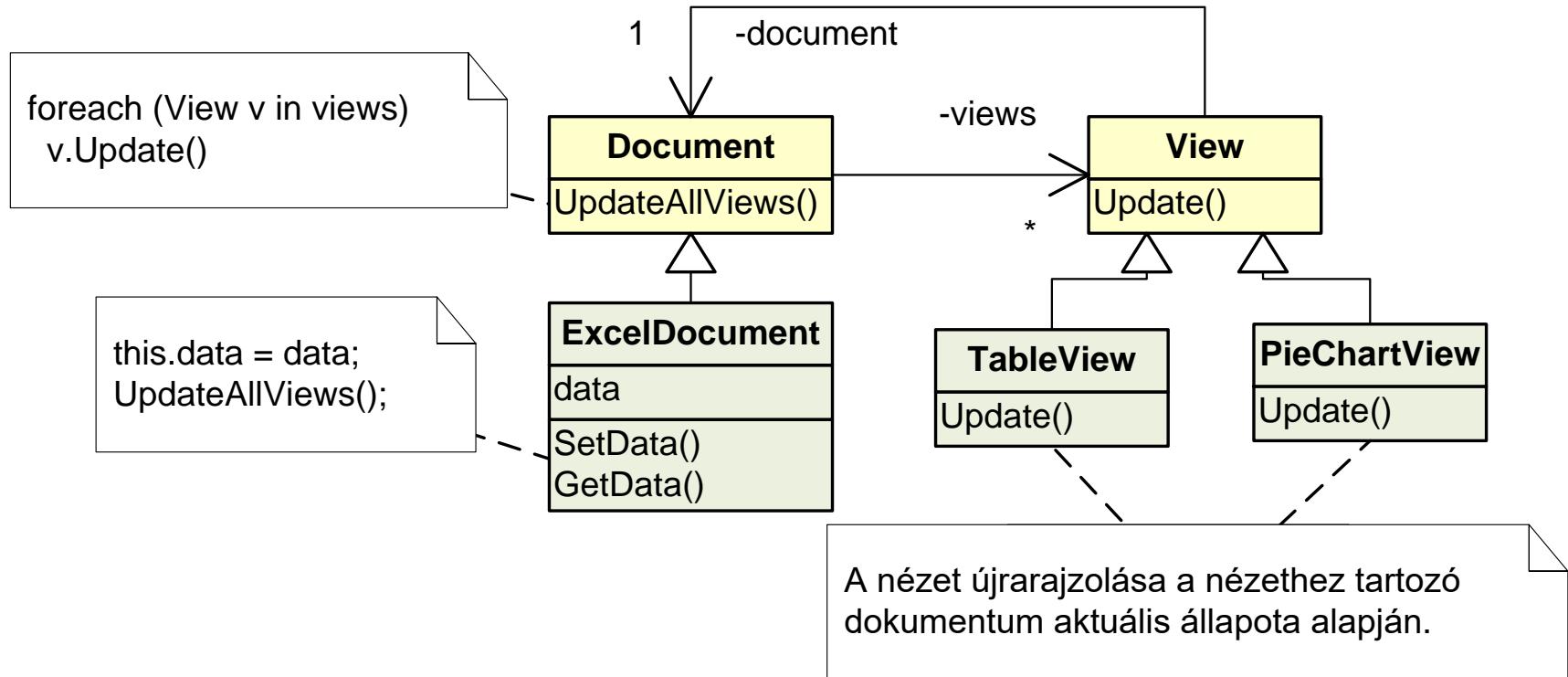
- A jó megoldás

- > Emeljük ki az adatokat és az azon értelmezett műveleteket egy osztályba, ez lesz a dokumentum (vagy modell)
- > A dokumentumhoz különböző view-kat lehet beregisztrálni
- > Ha valamelyik view megváltoztatja a dokumentum adatait, a dokumentum értesíti az összes beregisztrált view-t a változásról.
- > Az értesítés hatására a view lekérdezi a dokumentum állapotát és frissíti magát
- > A dokumentum csak egy közös View interfészen/ősosztályon keresztül tárolja a beregisztrált view-kat (nem függ az egyes típusoktól).



Statikus nézet (osztálydiagram)

- Példa (Excel)



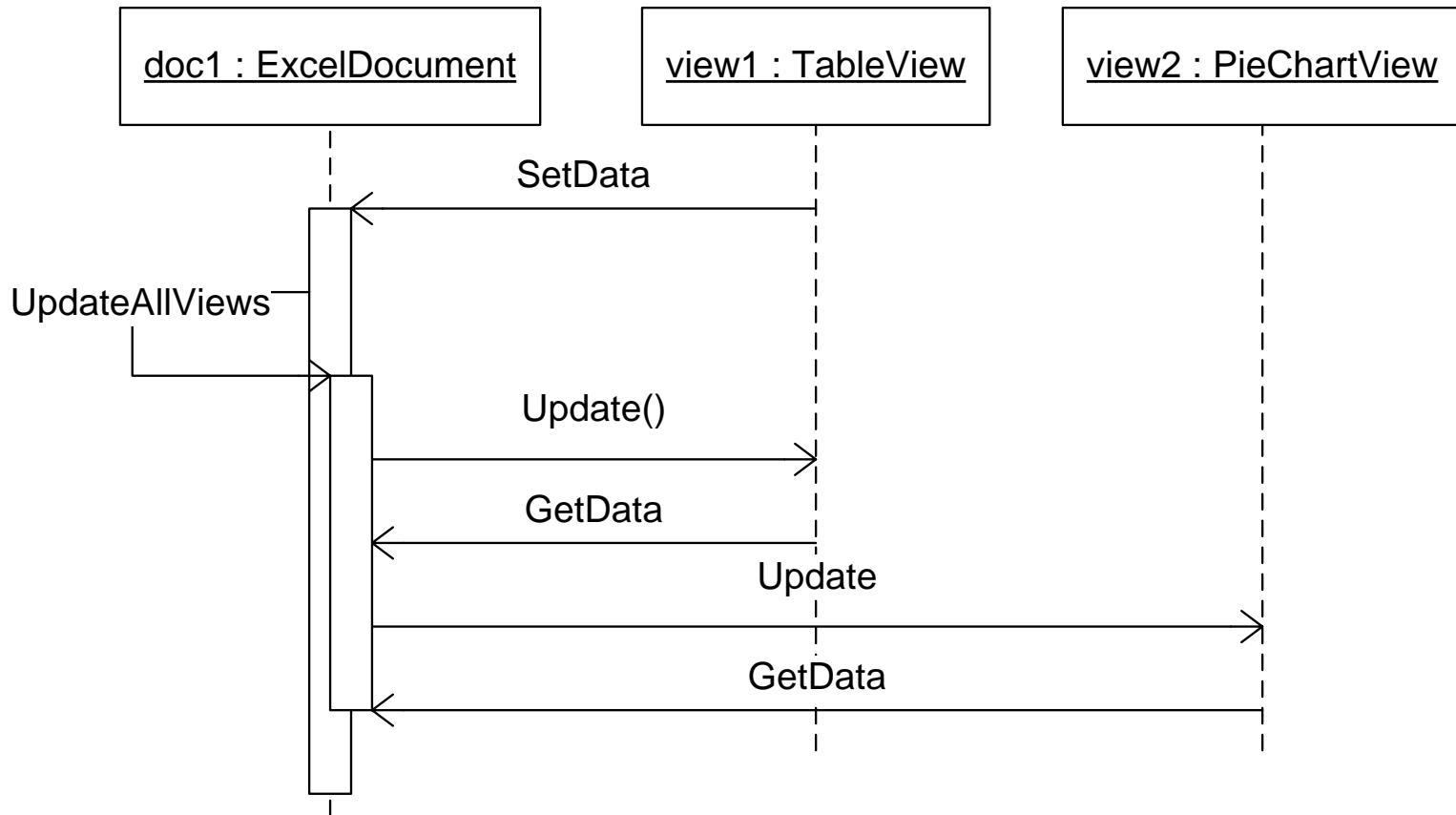
Osztálydiagram magyarázat

- Document
 - > Az osztály objektumai reprezentálnak egy-egy dokumentumot, és a views nevű listájában tárolja a beregisztrált nézeteket.
- ExcelDocument
 - > Egy példa Document leszármazottra.
 - > Tárolja a data, stb. tagváltozójában a dokumentum adatait (pl. cella értékek).
 - > Publikus lekérdező függvényeket biztosít a többi osztály számára az adatokhoz (elsősorban a nézet számára), pl. GetData.
 - > Publikus adatmódosító műveleteket biztosít a többi osztály számára, pl. SetData. Ezek módosítják a tagváltozókat, majd az UpdateAllViews hívásával értesítik a többi nézetet a változásról.
 - > Az UpdateAllViews frissíti a beregisztrált nézeteket (Update-et hív mindenre).

Osztálydiagram magyarázat

- View
 - > Az egyes nézetek közös ōse vagy interfésze, lehetővé teszi egységes kezelésüket.
 - > Tartalmaz egy referenciát a dokumentumra, amin keresztül a leszármazott nézetek elérhetik a dokumentumot, melynek adatait megjelenítik.
- TableView, PieChartView, stb.
 - > A dokumentum egyes nézeteit reprezentálják.
 - > A View-ból származnak.
 - > Felüldefiniálják vagy implementálják a View Update műveletét. Ebben a dokumentum aktuális állapota alapján frissítik a nézetet.
- Windows Forms esetben:
 - > Egy nézetet tipikusan egy Form leszármazott ablak vagy egy UserControl reprezentál.
 - > Az Update-ben tipikusan Invalidate hívás van, az újraraajzolás a Paint eseménykezelőben történik.

Dinamikus nézet (szekvenciadiagram)



Konklúzió

- Document-view előnyök
 - > A modell kódjában csak egy View lista van, így a modell független az egyes View-t implementáló osztályoktól.
 - > A modell újrafelhasználható!
 - > Egy egyszerű mechanizmust kaptunk arra, hogy az összes view konzisztens nézetét mutassa az adatoknak.
 - > A rendszer könnyen kiterjeszthető új view osztályokkal. Sem a modellt, sem a többi view osztályt nem kell ehhez módosítani.
- Document-view hátrányok
 - > Megnövekedett komplexitás

Az MVC architektúra

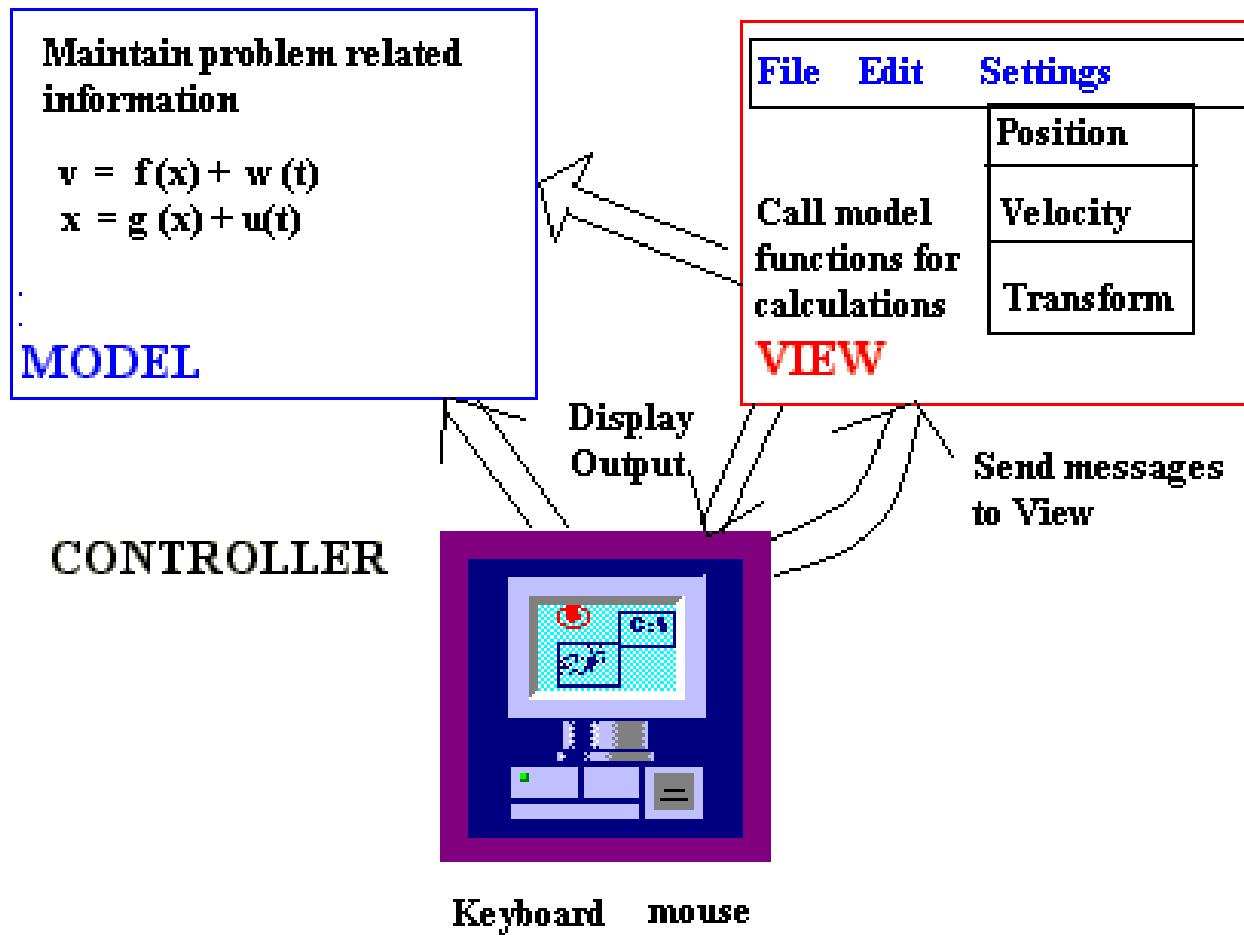
A Model-View-Controller architektúra

- Felhasználói felülettel rendelkező alkalmazások
- Régebbi, mint a Document-View
- A SmallTalk nyelvben jelent meg
- Trygve Reenskaug

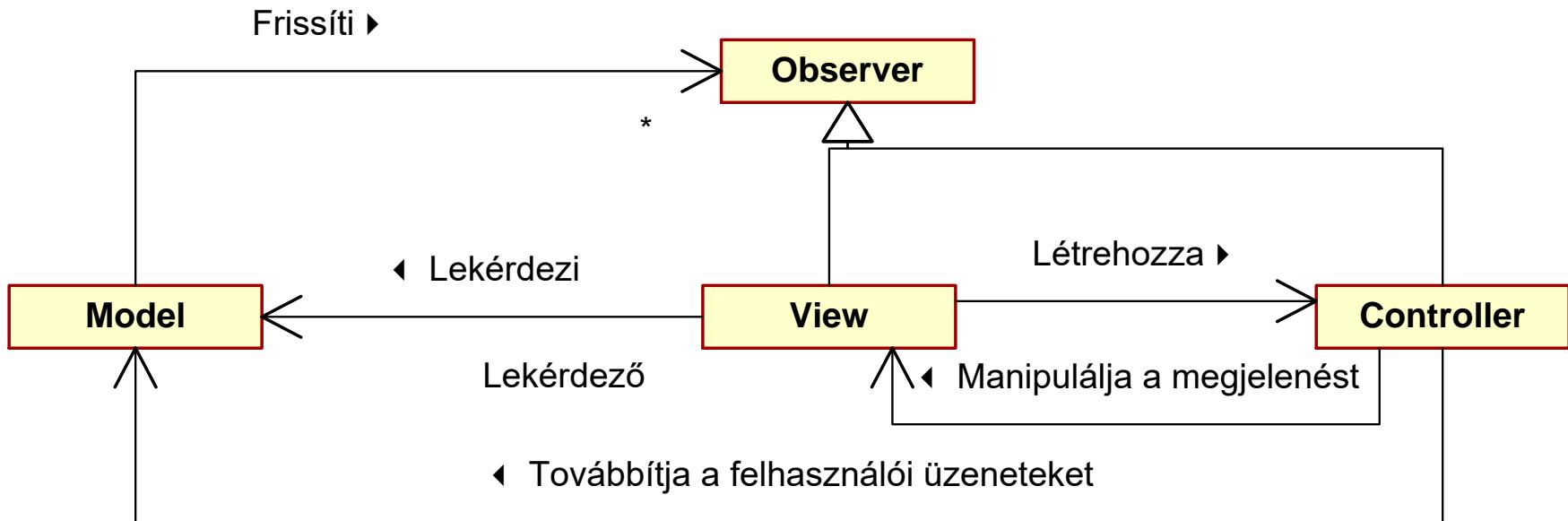
Az MVC architektúra

- Alapigazság: ne drótozzuk bele a GUI-ba az alkalmazáslogikát
- **MVC esetén**
 - > Model – alkalmazáslogika
 - Doc-View: Document
 - > View – megjelenítés
 - Doc-View: View
 - > Controller – interakció: kommunikáció a felhasználóval
 - Doc-View: View

Az MVC architektúra



MVC osztálydiagram



MVC osztálydiagram

- Model
 - > Tartalmazza az adatokat
 - (tagváltozók formájában),
 - > Valamint olyan műveleteket, melyek ezeken az adatokon dolgoznak
 - (pl. Save, Clear, stb).
- View
 - > Megjeleníti az adatokat.
 - > Az adatokat a modelből olvassa ki (nem tárolja ő is).
 - > Tartalmazhat olyan adatot, ami csak a nézetre vonatkozik
 - (pl. Zoom mértéke).

MVC osztálydiagram

- Controller
 - > A felhasználói interakciókat kezeli.
 - > Itt vannak a billentyűzet-, egér- és menüeseményeket kezelő függvények, melyek tipikusan a modelbe hívnak bele, vagy esetleg egy viewba (ha az adott művelet a viewra vonatkozik, pl. Zoom változtatása).
- Observer
 - > ha egy controller megváltoztatja a modelt, akkor a model valamennyi viewt és controllert értesít, hogy frissítsék magukat a modellből.
 - > Ez garantálja, hogy minden view konzisztens nézetét mutatja az adatoknak, és a contollerek is tudják tiltani/engedélyezni a felhasználói parancsok futtatását
 - (pl. File/Close menü tiltandó, ha nincs megnyitott dokumentum).
 - > A model egy közös Observer típusú listában tárolja a viewkat és controllereket.

Időbeli működés

- Működés
 - > A Controller kezeli az eseményeket
 - > A Controller értesíti az eseményről a Modelt
 - > A Model értesíti megváltozásáról a Viewt/Viewkat
 - > A View(k) lekérdezi(k) a Model állapotát, majd megjeleníti(k) azt
- Inicializálás (nem kell tudni)
 - > A Főprogram létrehozza a Model-t
 - > A Főprogram létrehozza a View-t
 - > A View létrehozza a Controller-t
 - > A View magához csatlakoztatja a Model-t

Az MVC értékelése

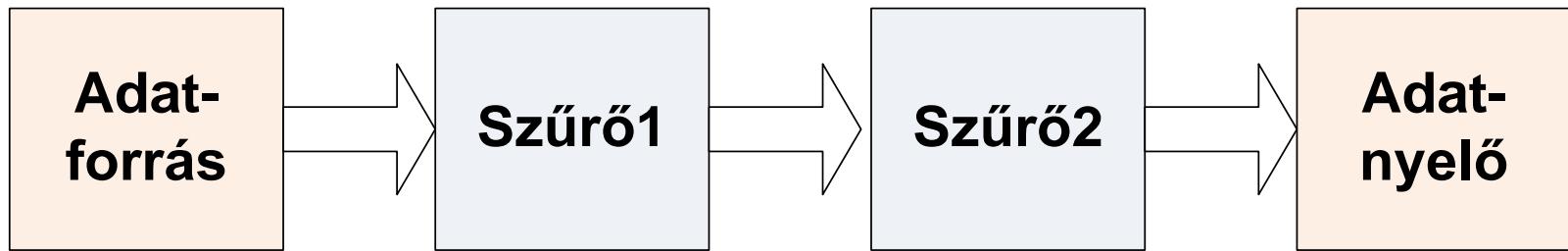
- Előnyök
 - > Többféle nézete ugyanannak az adatnak
 - > Szinkronizált nézetek
 - > Függetlenül cserélhető, kibővíthető View-k és Controller-ek
 - > A modell könnyebben tesztelhető (pl. NUnit)
 - > Lehetséges framework
- Hátrányok
 - > Komplexitás növekszik
 - > Túl sok szükségtelen frissítés lehet
 - > A View és a Controller gyakran nem különválasztható, nem is célszerű. Megoldás a Controller és a View összevonása:
 - Document-View architektúra (MFC)
 - Az MVC inkább a webes világban terjedt el.

Csővezetékek és szűrők (Pipes and Filters)

Csővezetékek és szűrők

- Architekturális minta, amely struktúrát ad **adatfolyamot** feldolgozó rendszerekre
- minden feldolgozási lépést egy **szűrő** végez el
- Az egyes szűrőket **csővezetékek** kötik össze
- A szűrők többféleképpen is **kombinálhatók**, így több rendszer is kialakítható

Struktúra



Példák

- Videofolyam feldolgozás
 - > Kontrasztállítás
 - > Világosságállítás
 - > Színmélység-állítás
 - > Átméretezés
 - > Formátumváltás, tömörítés
- Banki elektronikus jelentések feldolgozása
(adattisztítás, normalizálás, ... lépések)

Csővezetékek és szűrők

- Hálózata a következő komponensek:
 - > Adatforrás
 - > Adatnyelő
 - > Szűrők
 - > Csővezetékek

Szűrők

- Transzformációt hajtanak végre
 - > Bemenet egy csővezeték
 - > Kimenet egy csővezeték
- Aktív szűrő
 - > Ciklusban várakozik
 - > Ha a bemenő csővezetéken adat van, behúzza
 - > A kimeneti csővezetékre kitolja az adatot
- Passzív szűrő
 - > A bemenetén a csővezeték tolja be az adatot
 - > A kimenetén a csővezeték húzza ki az adatot

Csővezetékek

- Átviszik az adatot
- Bufferelhetik az adatot*: FIFO
- Szinkronizálhatnak az aktív szűrők között
 - > (csak ha aktív a szűrő)

Forgatókönyvek

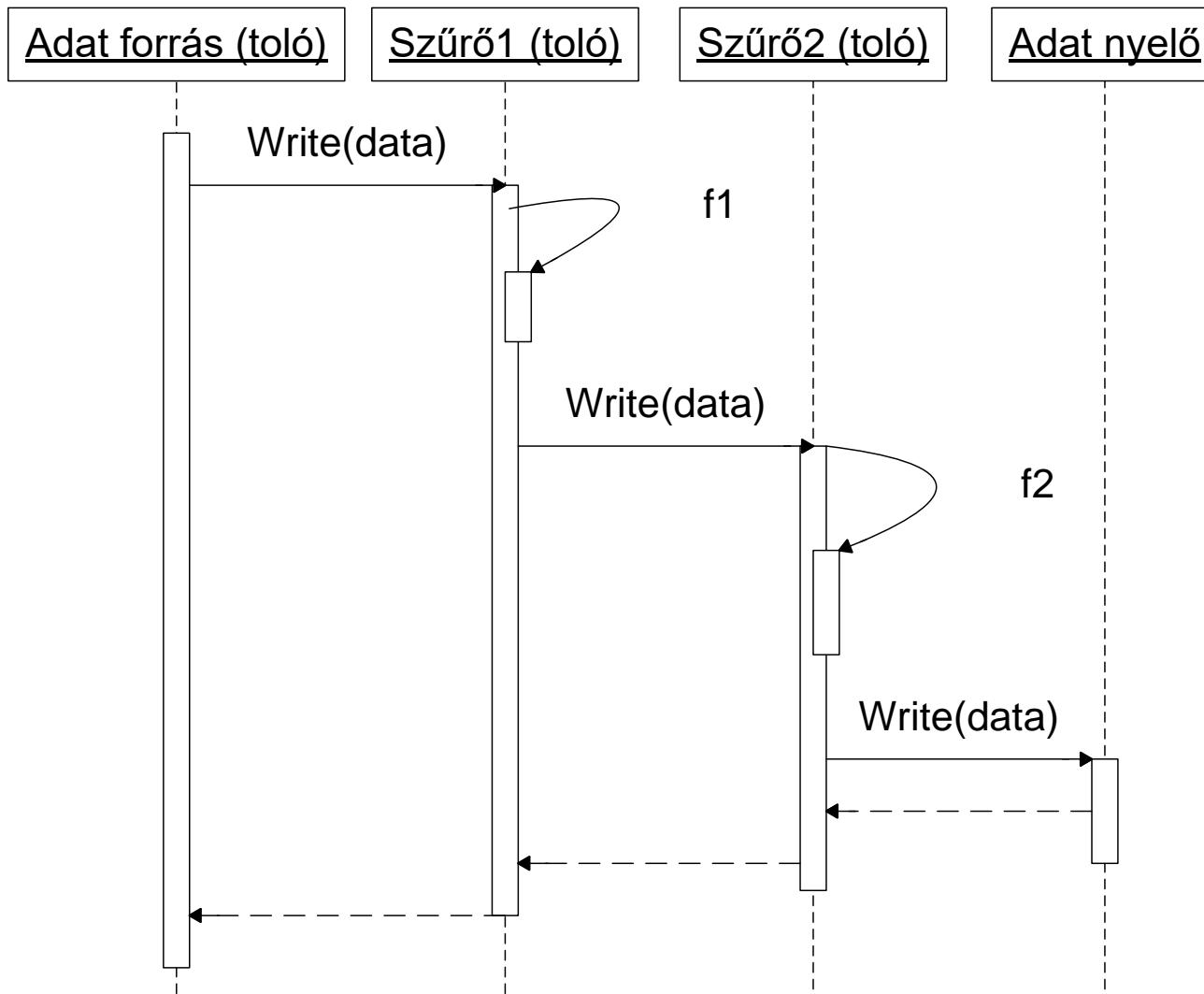
- Passzív szűrők
 - > Adatforrás által vezérelt
 - > Adatnyelő által vezérelt
- Aktív szűrők
 - > Egy aktív szűrő által vezérelt
 - > Aktív szűrők által vezérelt

Adatforrás által vezérelt (passzív szűrők)

- Filter pszeudokód:

```
void Write(Data data)
{
    Data processedData = ProcessData(data);
    nextFilter.Write(processedData);
}
```

Adatforrás által vezérelt

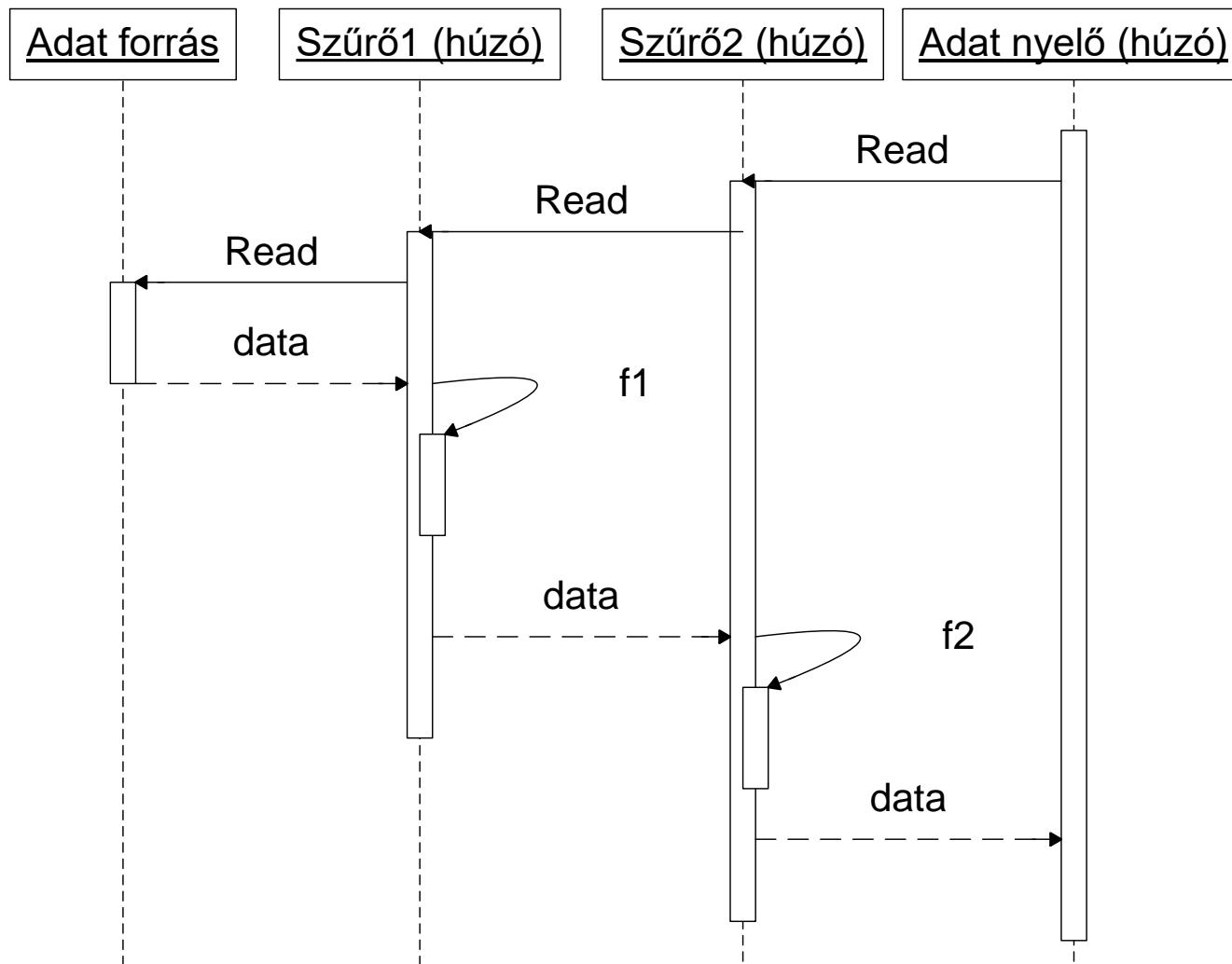


Adatnyelő által vezérelt

- Filter pszeudokód:

```
Data Read()
{
    Data data = prevFilter.Read();
    Data processedData = ProcessData(data);
    return processedData;
}
```

Adatnyelő által vezérelt



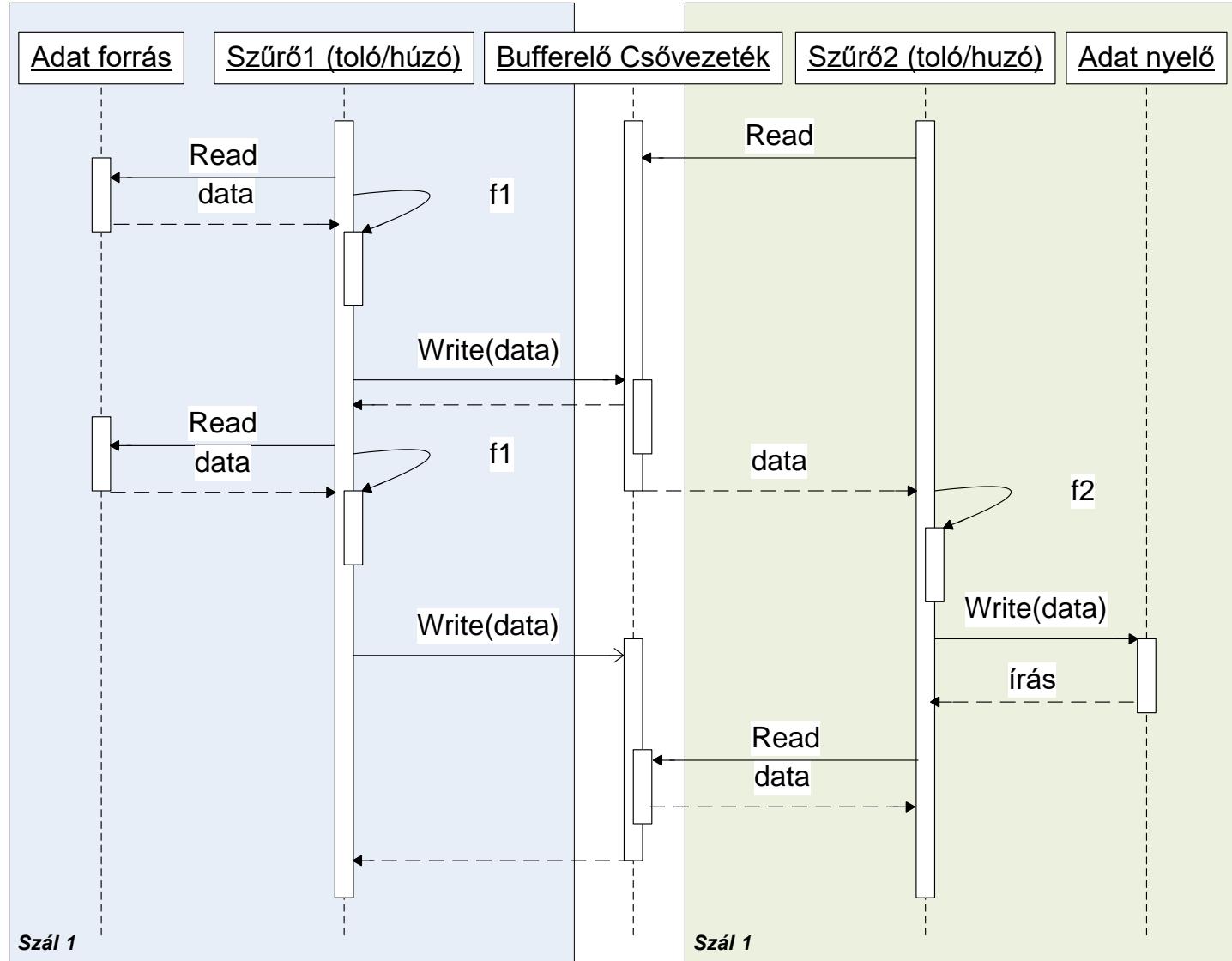
Forgatókönyvek

- Aktív szűrők
 - > Mindegyik szűrő párhuzamosan fut
 - > Mindegyik szűrő várakozik a bementén megjelenő adatokra
 - > Az adat kitolás nem blokkolt, csak ha tele van a kimeneti pipe
- Filter pszeudokód:

```
void Run()
{
    Data data;
    while (data = inputPipe.Read())
    {
        Data processedData = ProcessData(data);
        outputPipe.Write(processedData);
    }
}
```

- További forgatókönyvek is léteznek!

Több aktív szűrő



Előnyök

- A filterek tetszőlegesen kombinálhatók
- Az egyes filterek lecserélhetők
- A filterek újrafelhasználhatók
- Párhuzamos feldolgozás lehetősége (aktív szűrők esetén)
- Gyors prototípus készítés meglevő filterekkel
 - > pl. Unix sed és awk.
 - > A teljesítménykritikusak majd hatékonyabbra cserélhetők
- Lehetőség közbelső fájlok használatára (de nem kötelező)

Nehézségek

- Hibakezelés
 - > Ha hiba van, mi történjen, honnan folytassuk.
- Adattranszformációs overhead
 - > Pl. Unix pipe – lebegőpontos számokat ASCII-be kell konvertálni és vissza minden csővezeték esetében
 - Ugyanis sortörés „darabolja” az adatfolyamot

Irodalom

- Martin Fowler: Analysis Patterns (két és háromrétegű architektúra bemutatása)
- Frank Buschmann, ...: Pattern-Oriented Software Architecture Volume 1: A System of Patterns

Ellenőrző kérdések I.

- Magyarázza el röviden a szoftver architektúrájának a fogalmát! Hol a leghangsúlyosabb az architekturális tervezés a RUP módszertanban?
- Ismertesse a rétegelt architektúra alapkoncepcióit és leggyakoribb forgatókönyveit egy-egy példával!
- Ismertesse a rétegelt architektúra előnyeit!
- Ismertesse a vállalati információs kétrétegű architektúráját! Adja meg az architektúra előnyeit és hátrányait!
- Ismertesse a vállalati információs háromrétegű architektúráját! Adja meg az architektúra előnyeit és hátrányait!

Ellenőrző kérdések II.

- Ismertesse a csővezetékek és szűrők architektúrát, és adjon egy példát annak alkalmazására!
- Mutasson be három lehetséges forgatókönyvet a „csővezetékek és szűrők architektúra” esetén!
- Ismertesse az MVC architektúrát! Mutassa be a minta szereplőit, adja meg a minta osztálydiagramját, valamint ismertesse a minta működését!
- Ismertesse a Dokumentum/Nézet architektúrát! Mutassa be a minta szereplőit, adja meg a minta osztálydiagramját, valamint ismertesse a minta működését szekvenciadiagrammal!

Tervezési minták (design patterns)

Szoftvertechnikák

Benedek Zoltán



Automatizálási és
Alkalmazott
Informatikai Tanszék

Tervezési minták

- > Definíció
- > Áttekintés
- > Szervezésük
- > Miben segítenek a tervezési minták
- > Néhány fontosabb, érdekesebb minta részletesebb tárgyalása

A Java kapcsán már volt

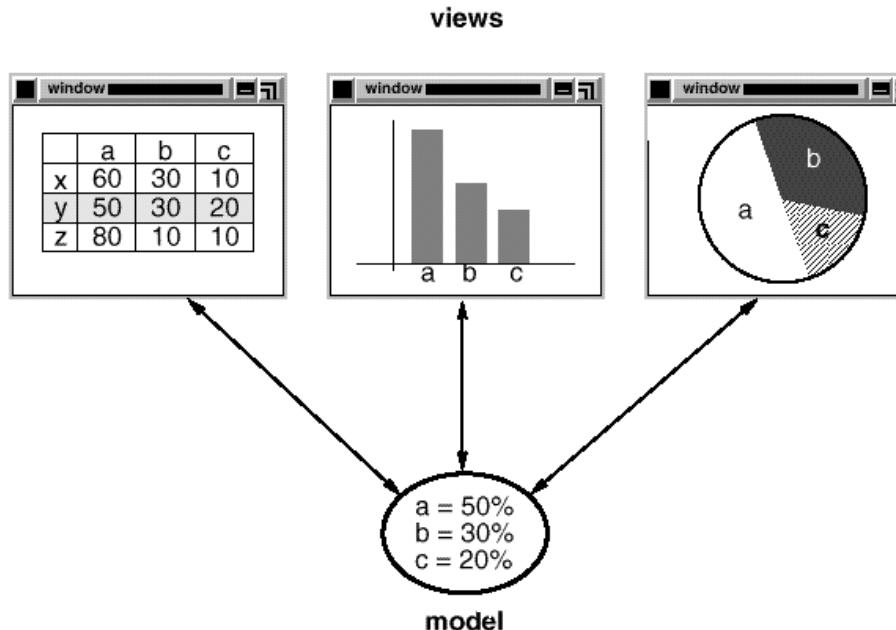
- > *De most már alkalmazzuk mi is*

Definíció

- Tervezési minta leír egy gyakran előforduló problémát, annak környezetét és a megoldás magját, amit alkalmazva számos gyakorlati eset hatékonyan megoldható.
- Példa: MVC architektúra
- Definíció - mit értünk tervezési minta alatt
 - > Szorosabb definíció: most ilyen értelemben fogjuk használni
 - > Lazább: pl. az interfész is “egyfajta” pattern

Példa - Model-View-Controller (MVC) architektúra

- Smalltalk-80 alatt user interfészek fejlesztésére
- Három objektum szereplő
 - > Model – applikáció objektum
 - > View – megjelenítésért felelős
 - > Controller – definiálja, hogy a user interfész hogyan regál a felhasználói bemenetekre
- Tekintsük csak a view és a model viszonyát



Példa - Model-View-Controller (MVC) architektúra

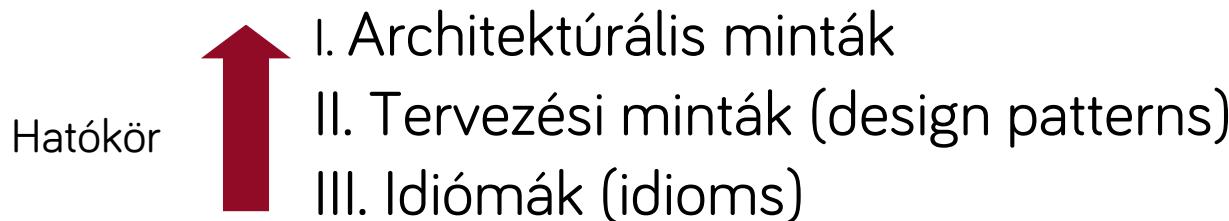
- Az adat (model) és a megjelenítés (view) ketté van választva, és a kettő között előfizetés/értesítés viszony van
 - > Egy model-hez több view-t is lehet csatolni, minden egyik view a model egy adott nézetét jeleníti meg
 - > Garantálni kell, hogy minden egyik view a model aktuális állapotát tükrözze. Amikor a model megváltozik, akkor valamennyi view-t sorra értesíti a változásról, a view-k lekérdezve a model állapotát frissítik magukat
 - > A model megváltoztatása nélkül lehet új view-t a modellhez kapcsolni
- Általánosítva, elvonatkoztatva a model-view esettől
 - > A fenti megközelítés lehetővé teszi, hogy egy objektum megváltozása esetén értesíteni tudjon tetszőleges más objektumokat anélkül, hogy bármit is tudna róluk.
 - > Ez az ún. observer pattern

Tervezési minták leírása

- A tervezési minta leírásához négy alapelem tartozik
 - > Pattern név (pattern name)
 - A pattern neve, hatékonyan azonosítja a mintát.
 - **Fontos a kommunikáció miatt:** a fejlesztőknek elég a pattern nevére hivatkozniuk, ebből már értik is, hogy milyen objektumok milyen szerepkörben szerepelnek az adott tervben/kódban (persze csak ha ismerik az adott mintát)
 - > Probléma (problem)
 - A **probléma** és a **környezet (context)** bemutatása, gyakran konkrét példán keresztül
 - > Megoldás (solution)
 - Leírja a megoldásban szereplő elemeket, kapcsolatukat, az egyes elemek felelősségeit és együttműködését. **Nem** konkrét eset megoldása, inkább a megoldás absztrakt leírása.
 - > Következmények (consequences)
 - A minta alkalmazásának következményeit írja le. Fontos tapasztalatokat tartalmaz, és segít a minta kiválasztásban.

Minták csoportosítása hatókör szerint

- A szoftverrendszer milyen szintjén használhatók (architektúra → alrendszer → ... → kódolás egy adott programnyelven)
- Három csoport



Minták csoportosítása hatókör szerint

- I. Arhitektúrális minták - előző előadások témája volt
 - > **Alapstruktúrát adnak a rendszernek.**
 - > A szoftverrendszer (egyszerű esetben alkalmazás) alapvető struktúráját, szervezését adják meg.
 - > Alrendszereket definiálnak, meghatározzák az alrendszerök felelősségeit, valamint szabályokat és irányelveket adnak meg a kapcsolatukra.
 - > Pl.:
 - Rétegek
 - Model-View-Controller architektúra
 - dokumentum alapú ablakos rendszerekben
 - Document-view architektúra
 - dokumentum alapú ablakos rendszerekben
 - Pipe-ok
 - stream feldolgozás
 - akár tetszőleges sorrendben kombinálhatók az egyes komponensek
 - pl. szűrők alkalmazása
 - lásd: UNIX
 - Stb.

Minták csoportosítása hatókör szerint

- II. Tervezési minták
 - > Ezek a hagyományos értelemben vett tervezési minták
 - > Alrendszeren, szoftver komponenseken belül használhatók
 - Hatókör szempontjából középen helyezkednek el
 - A rendszer alapvető struktúrájára nincsenek hatással
 - Programozási nyelvektől viszont függetlenek

„Egy tervezési minta egymással kommunikáló komponensek egy gyakran ismétlődő struktúráját írja le, amely megold egy általános tervezési problémát egy adott környezetben.”

E. Gamma R. Helm R. Johnson J. Vlissides: Design Patterns

Minták csoportosítása hatókör szerint

- III. Idiomák
 - > Alacsony szintű tervezési minták.
 - > Egészen speciálisak, általában egy adott programozási nyelvhez kötődnek.
 - > Egy adott programozási nyelv esetében a nyelv eszközeivel egy gyakran felmerülő problémát hogyan lehet általánosan és hatékonyan megoldani.
 - > Pl. C++ -ban: Smart Pointer
 - Környezet
 - Egy objektumot több osztály kezel.
 - Probléma
 - Hatékonyan csak úgy lehet, ha pointert vagy referenciát használunk, vagyis az adott objektumot megosztottan használjuk. Ekkor viszont ki a felelős az objektum felszabadításáért (memory leak veszély)?
 - Megoldás
 - Definiálunk egy smart pointer (vagy handle) osztályt. Ezt ugyanúgy lehet használni, mint egy közönséges pointert, de egy plusz szolgáltatást is nyújt. A tényleges objektumot csak a hozzá tartozó smart pointeren keresztül lehet elérni. A smart pointer objektum számolja a referenciákat a hozzá tartozó objektumra. Ha a referenciák száma elérte a nullát, felszabadítja a hozzá tartozó objektumot.
 - Implementáció
 - C++ -ban template osztállyal célszerű



Minták csoportosítása hatókör szerint

- Idiomák – smart pointer példa (*NEM KELL TUDNI A KÓDOT*)
 - > a, ha “lemarad” a delete, elszivárog a memória
 - > b, itt az auto_ptr smart pointer osztály gondoskodik a referencia számlálásról, és szükség esetén felszabadítja a MyClass osztály objektumát (vagyis nincs szükség a delete meghívására, mint az a main függvényben látható)

```
int main(int argc, char* argv[])
{
    MyClass* p = new MyClass;
    p->DoSomething();
    delete p;
}
```

```
// smart pointer template osztály RÉSZLET (a
// referencia számlálás nincs jelölve)
template <class T>
class auto_ptr {
    T* ptr; // mutató az objektumra
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr() { delete ptr; }
    T& operator*() { return *ptr; }
    T* operator->() { return ptr; }
    // ...
};

int main(int argc, char* argv[])
{
    auto_ptr<MyClass> p(new MyClass);
    p->DoSomething();
}
```

Minták csoportosítása hatókör szerint

- Mostantól a középső szintre besorolható tervezési mintákról, vagyis a design pattern-ekről lesz szó
- Ezt nem kell túl szigorúan venni, több design pattern jól alkalmazható architektúra vagy idióma szinten is

GoF tervezési minták kategóriái (nem kell tudni)

		Cél		
		Létrehozási (creational)	Strukturális (structural)	Viselkedési (behavioral)
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Leírásuk

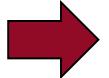
- A design patternek nem rendelkeznek önálló leíró nyelvvel
 - > OMT vagy UML jelölésrendszerrel
 - Osztálydiagram
 - Szekvenciadiagram
 - > Szöveges leírás

Pattern katalógus

- A katalógusban a patternek leírásának formátuma (**NEM KELL TUDNI**)
 - > **Name**: pattern neve
 - > **Intent**: a pattern használatának célja, mit valósít meg a pattern
 - > **Also known as**: alternatív nevek
 - > **Motivation**: gyakran gyakorlati példán keresztül bemutatja, hogy milyen tervezési problémát és hogyan old meg a pattern
 - > **Applicability**: környezetek leírása, ahol a pattern jól alkalmazható
 - > **Structure**: a megoldás osztály és objektum vázlata
 - > **Participants**: a patternben szereplő osztályok, objektumok rövid leírása
 - > **Collaboration**: az egyes elemek együttműködésének leírása
 - > **Consequences**: értékelés, a pattern milyen eszközökkel, milyen hatékonyan oldja meg az adott feladatot
 - > **Implementation**: a pattern implementációs problémáinak leírása, jó tanácsok, rövid kód részletek, nyelvfüggő implementációs kérdések
 - > **Sample code**: jellegzetes kód részletek, amelyek egy konkrét példán keresztül bemutatják a pattern alkalmazását
 - > **Known uses**: létező ismertebb rendszerek megemlítése, ahol a pattern fellelhető
 - > **Related patterns**: szorosan kapcsolatban álló patternek említése

Hogyan segítenek a probléma megoldásában

- A tervezési minták a fejlesztés tervezés fázisában segítenek (az analízis minták az analízis fázisában)
- Nehéz (ezekben segítenek a patternek):
 - > I. Megfelelő objektumokat megtalálni, definiálni
 - Objektumok számának, méretének meghatározása
 - Objektum interfészek definiálása
 - Objektum implementálása
 - > II. Újrafelhasználható kódot készíteni (design for reuse)
 - > III. Könnyen változtatható, kiterjeszhető kódot készíteni (design for change)

Részletesebben 

Hogyan segítenek a probléma megoldásában

- Objektumok megtalálása nehéz
 - > Objektum orientált tervezési eljárások igyekeznek a problémát megoldani: természetes nyelven írt specifikáció alapján például a főnevek és a igék kigyűjtése segíthet az osztály illetve az objektum hierarchia kialakításában.
 - > De a rendszerek számos olyan osztályt tartalmaznak, amelyeknek nincs megfelelője a valós világban, de a rendszerben fontos szerepet játszanak:
 - Bizonyos alacsony szintű osztályok (pl. a tömb osztály)
 - Control osztályok (a boundary és entity általában egyszerűbb)
 - ...
 - > Design patternek segítenek megtalálni a kevésbé magától értetődő osztályokat

Újrafelhasználhatóság (reuse)

- Nem adódik magától, szem előtt kell tartani a tervezéskor. A tervezés egyik célja!
 - > A költségcsökkentés leghatékonyabb módja
 - > Hosszú távon kifizetődő lehet: saját keretrendszer kifejlesztése
- Kitérő: Keretrendszer (framework) definíció
 - > Adott probléma területen (pl. ablakozós rendszerek, dokumentumkezelő rendszerek, matematikai rendszerek, grafikus rendszerek, CAD rendszerek, stb.)
 - Meghatározza az alkalmazás architektúráját, ehhez alapvető építőköveket (pl. osztályok) biztosít.
 - OO környezetben: előre definiált, együttműködő osztályok vannak, pontosan meghatározott felelősséggel. Tipikusan
 - le kell belőlük származtatni,
 - virtuális függvényeket kell felüldefiniálni
(Pl. Windows Forms –ban Form ősosztály)
 - Cél: A lehető legkevesebbet kelljen kódolni.
 - Konzisztens struktúrája lesz az alkalmazásainknak
 - Magas szinten már “tervezni sem kell”, hiszen az architektúra meghatározott - terv újrafelhasználás (design reuse)

Változtathatóság

- Nem adódik magától, szem előtt kell tartani a tervezéskor.
- A tervezés egyik célja!
- Miért?
 - > Rendszert tervezni nehéz
 - > Sokszor nem is tudjuk az elején pontosan definiálni a feladatot
 - > A szoftver termékek az első kiadás után általában hosszú éveken át, számos iterációban fejlődnek tovább

Változtathatóság

- Tervezzünk úgy, hogy a rendszer nem végleges (design for change)!
 - > Ha változtatni kell, ne kelljen kidobni a meglevő részeket,
 - > Ha változtatni kell,
 - Minél kevesebb helyen kelljen változtatni
 - A meglevő részek minél kisebb változtatásával (pl. csak le kelljen származtatni, ekkor változtatni sem kell)
 - Így gyorsabban haladunk és kevesebb új bugot viszünk a meglevő részekbe!
 - > Nem minden, mert költséges! Gondolkozzunk előre, mik a kritikus részek és mennyi bővítés várható!

Alapelv

- Cél: a függőségek csökkentése az egyes alrendszerek, komponensek között
 - > Egy változtatás minél kevesebb komponenst érintsen
 - Változtathatóság
 - Újrafelhasználhatóság

System of Patterns

- Mi a system of patterns?
 - > A rendszer tervezésekor több mintát is felhasználunk
- Általában egy (illetve alrendszerenként egy) architektúra szintűt
 - > Ez megadja az alapstruktúrát
- Az egyes részek megtervezésekor (detailed design) több tervezési mintát is felhasználunk, akár egymással kombinálva
- A kombinált megoldásba minden minta beleviszi a maga pozitív tulajdonságait, vagyis egy megoldást egy részproblémára

Néhány fontosabb, gyakrabban használt tervezési minta

- Létrehozási
 - > Factory method
 - > Abstract factory
 - > Singleton
- Strukturális és viselkedési
 - > Observer
 - > Command
 - > Command Processor
 - > Memento
 - > Adapter
 - > Facade
 - > Proxy
 - > Composite
 - > Template method
 - > Strategy

Mit kell tudni?

A minta neve alapján vagy példán, vagy általánosságában ismertetni a mintát (milyen környezetben, milyen problémát, hogyan old meg) *. A bemutatáshoz a legtöbb mintához osztály, illetve néhány esetben szekvencia diagramot is kell rajzolni).

* MSc felvételin általánosságában is tudni kell ismertetni a mintákat, nem elég példán keresztül!

LÉTREHOZÁSI MINTÁK

Factory method

Abstract factory

Singleton

Factory Method

(gyártó metódus)

Factory Method

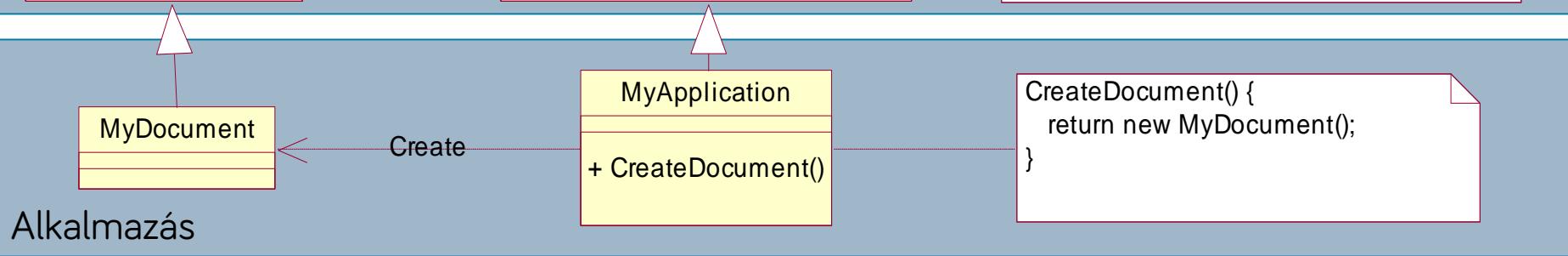
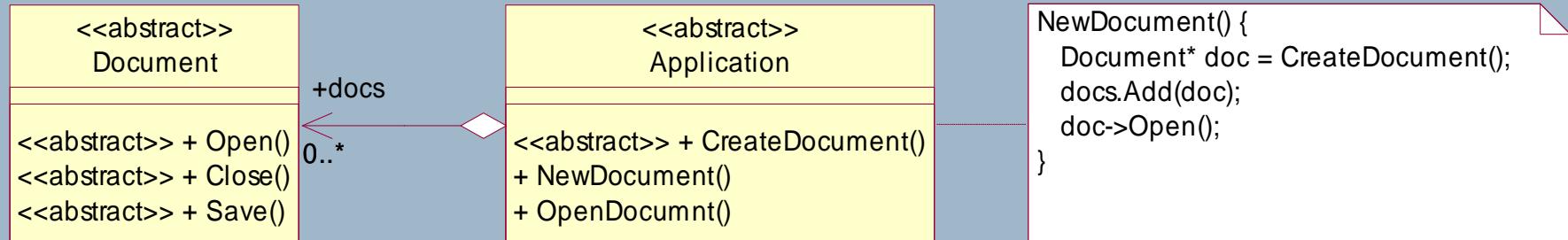
- Célja
 - > Interfész definiál az objektum létrehozására, de a leszármaztatott osztályra hagyja annak eldöntését, hogy konkrétan melyik osztályból kell példányt létrehozni.
 - > A Factory Method lehetővé teszi, hogy az új példány létrehozását az leszármazott osztályra bízzuk.
- Szokás virtuális konstruktornak is nevezni

Factory Method példa

- Framework, ami egyszerre több dokumentum kezelését támogatja (mint pl. a Visual Studio)
- A framework két, kulcsfontosságú osztálya:
 - > Application
 - > Document
- Természetesen mindenkitő absztrakt osztály, a programozónak kell leszármaztatnia ebből a két osztályból, hogy megvalósítsa a céljainak megfelelő konkrét applikáció és dokumentum osztályokat (nevezzük ezeket MyApplication és MyDocument osztályoknak)
- A framework Application osztálya nem tudja, hogy milyen konkrét dokumentum osztályt kell létrehozni (hiszen a framework megírásakor a MyDocument osztály még nem létezett), csak azt tudja, hogy mikor kell létrehozni.

Factory Method

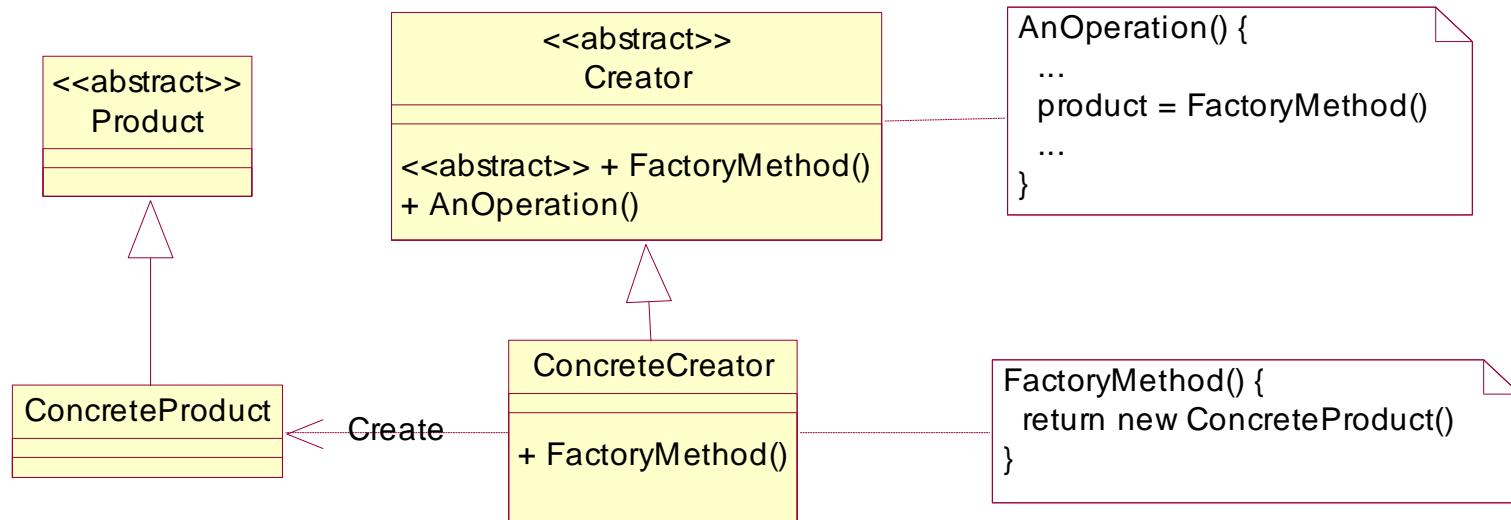
Keretrendszer



- Amikor a felhasználó kiválasztja a „New” menüelemet, a keretrendszer meghívja az **Application::NewDocument()**-tet.
- Ennek törzsébe nem írhatták be a framework készítői, hogy „new MyDocument()”, hiszen a framework megírásakor a **MyDocument** még nem létezett.
- Megoldás: az új dokumentum létrehozásához a **Application::NewDocument()** meghívja az **Application::CreateDocument()** absztrakt függvényt, amit az alkalmazásfejlesztőnek felül kell definiálnia úgy, hogy a **MyDocument** osztályból adjon vissza egy objektumot.

Factory Method

- Megoldás
 - > Factory method minta alkalmazása
 - > Bízzuk a leszármaztatott osztályra a konkrét objektum (dokumentum létrehozását)
- Struktúra



Factory Method

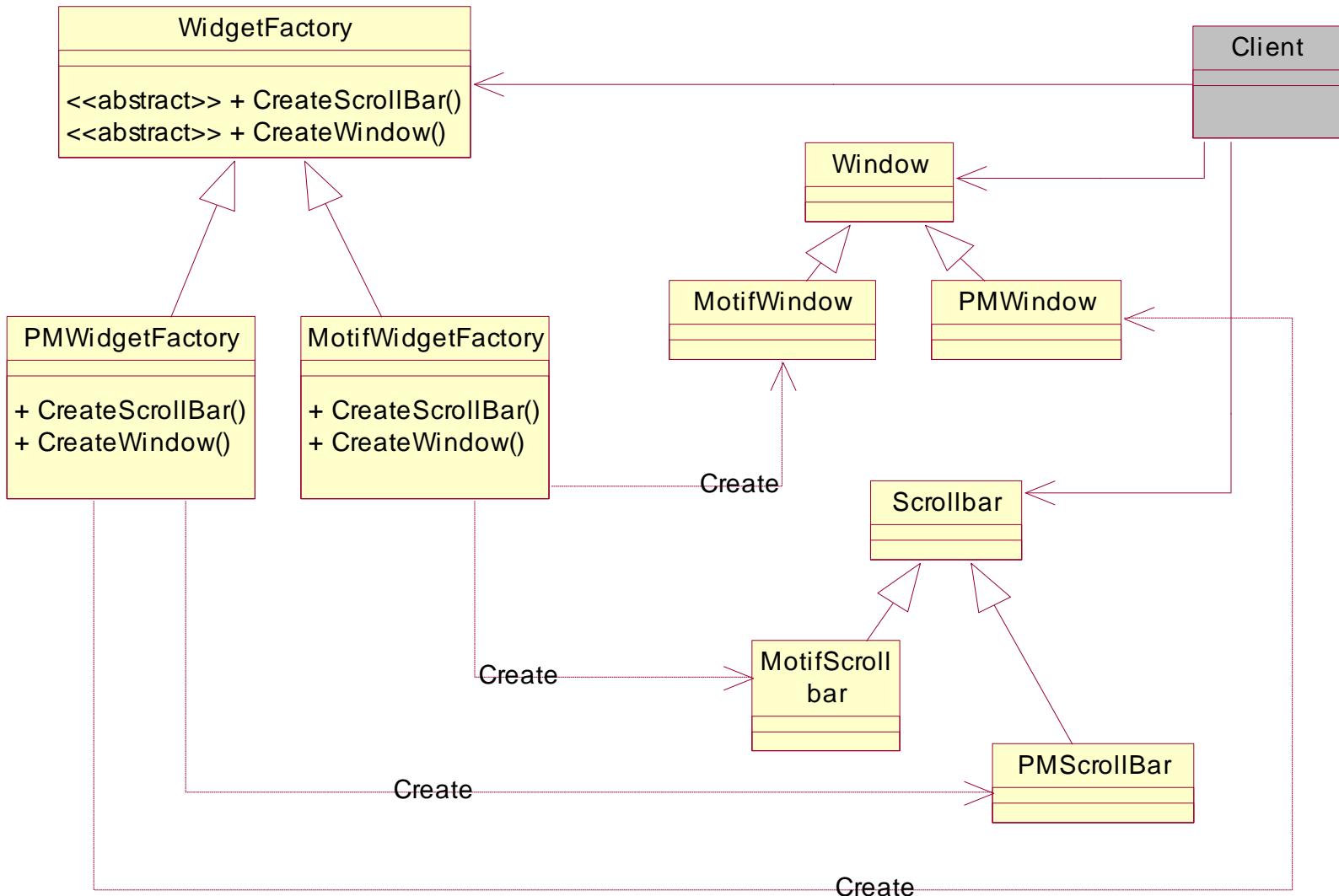
- Használjuk, ha
 - > Egy osztály nem látja előre annak az objektumnak az osztályát, amit létre kell hoznia
 - > Ha egy osztály azt szeretné, hogy leszármazottai határozzák meg azt az objektumot, amit létre kell hoznia

Abstract Factory

Abstract factory - példa

- Ablakos rendszerek, GUI vezérlőelemek (ablak, nyomógomb, kiválasztógomb, stb.)
- Több “look-and-feel” megjelenítést támogasson az alkalmazás
 - > Pl. Motif, Presentation Manager, Win7, Win10
- Hogyan lehet?
- minden GUI vezérlőelemnek külön verziót készítünk minden megjelenési formájához.
- Ne drótozzuk bele a felhasználói felület elemeket az alkalmazásba. Sem az elemek létrehozását, sem pedig a használatukat.
 - > Zárjuk egységbe ezek létrehozását, bízzuk más objektum(ok)ra
 - > Az alkalmazásban csak egy interfészen keresztül hivatkozzunk rájuk (így az implementációjuk kicserélhető)

Abstract factory



Abstract factory - Megoldás

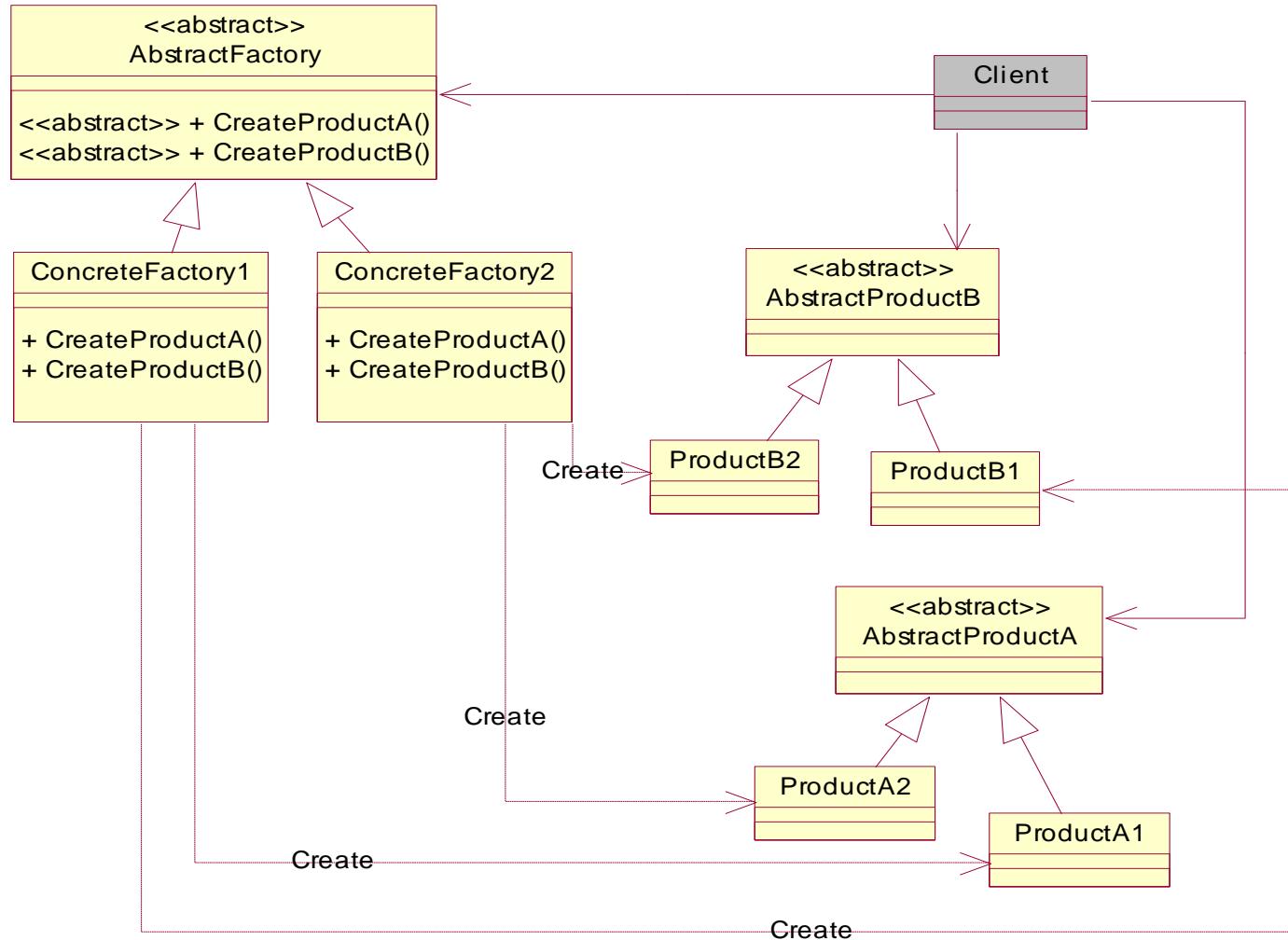
- Hozzunk létre egy absztrakt WidgetFactory osztályt, aminek az interfésze támogatja valamennyi vizuális elem létrehozását.
- Ennek a műveletei egy-egy vizuális elem objektummal térnek vissza (absztrakt vizuális elem osztályként - pl. Window és nem MotifWindow - hivatkozva)
- Mindegyik konkrét megjelenítéshez tartozik egy WidgetFactory-ból leszármaztatott osztály (MotifWidgetFactory, PMWidgetFactory), amelynek metódusai az adott megjelenítéshez tartozó felhasználói felületelem objektumokkal térnek vissza.
- A kliens az WidgetFactory interfészen keresztül egy konkrét WidgetFactory objektum (amivel fel volt konfigurálva előzetesen) megfelelő metódusának meghívásával hoz létre egy konkrét felhasználói felületelemet. Ezekre a konkrét felületelem objektumokra csak az absztrakt felületelem osztályokon keresztül hivatkozik.

(Így a kliens kódjában sehol nem szerepelnek konkrét Factory és felületelem osztályok, legfeljebb a felkonfigurálásnál, vagyis amikor megmondjuk, hogy Motif, vagy Presentation Manager elemekkel szeretnénk dolgozni. Ennek megfelelően a kliens teljesen független a konkrét felületelem osztályuktól)

Abstract factory

- Példa: lásd *AbstractFactory.cpp* C# vagy C++ forrás az előadásanyag melléklet zip-ben

Abstract factory - Struktúra



Gyakorlati példa 2

- **Feladat:** Tervezzünk adatkezelő független adatkapcsolati réteget többrétegű alkalmazásokhoz (pl. ADO.NET környezetben)
 - > ICommand interfész
 - SqlCommand
 - OracleCommand
 - OleDbCommand
 - > IConnection interfész
 - SqlConnection
 - OracleConnection
 - OleDbConnection
 - > ...
- A megoldás (abstract factoryval):
 - > IDbFactory interfész (CreateCommand, CreateConnection, stb, műveletekkel).
 - SqlDbFactory implementáció
 - OracleDbFactory implementáció
 - OleDbDbFactory implementáció

A .NET
beépítve
támogatja!

Abstract factory

- Használjuk, amikor
 - > A rendszernek függetlennek kell lennie az általa létrehozott dolgoktól (“termék” objektumok, pl. felhasználói felület elemek)
 - > A rendszernek több termékcsaláddal kell együttműködnie
 - > A rendszernek szorosan összetartozó “termék” objektumok adott családjával kell dolgoznia, és ezt akarjuk kényszeríteni a rendszerben (pl. Motif scrollbart ne lehessen Presentation Manager ablakkal együtt használni)

Abstract factory

- Előnyök
 - > Elszigeteli a konkrét osztályokat
 - > A termékcsaládokat könnyű kicserélni
 - > Elősegíti a termékek közötti konzisztenciát
- Hátrányok
 - > Nehéz új termék hozzáadása. Ekkor az Abstract Factory egész hierarchiáját módosítani kell, mert az interfész rögzíti a létrehozható termékeket
 - > Megjegyzés: ezt bizonyos esetekben ki lehet kerülni, pl. ha az előző példában minden termék egy közös GraphObject osztályból származik)

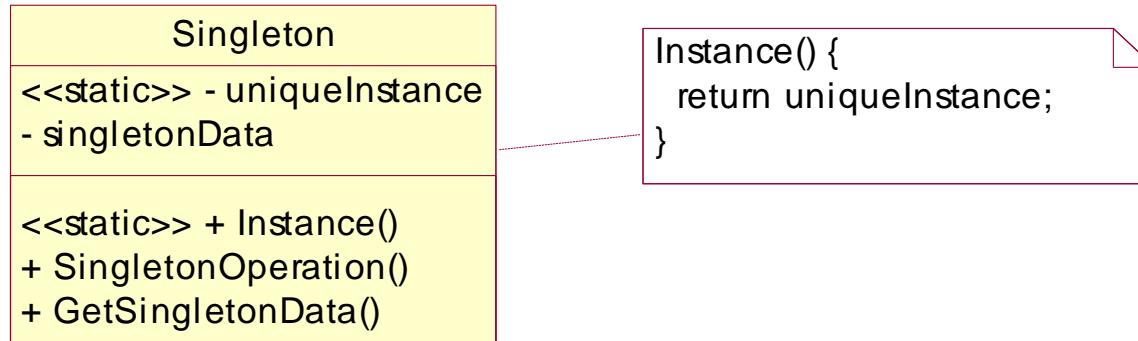
Singleton

Singleton

- Célja
 - > Biztosítja, hogy egy osztályból csak egy példányt lehessen létrehozni, és ehhez az egy példányhoz globális hozzáférést biztosít
- Elég gyakran van rá szükség
 - > Egy ablakkezelő objektum
 - > Egy fájlrendszer objektum
 - > ...
- Megoldás
 - > Singleton
 - Legyen az osztály felelőssége, hogy csak egy példányt lehessen belőle létrehozni
 - Biztosítson globális hozzáférést ehhez az egy példányhoz
 - > A programozási nyelvek segítségét igényli

Singleton

- Struktúra – itt nem ez a lényeg, a kód ismerete szükséges!



- Az `Instance` osztály-művelet (statikus!) meghívásával lehet példányt létrehozni, illetve az “egyetlen” példányt elérni. A `Singleton::Instance()`
- Mindig ugyanazt az objektumot adja vissza
- Bárhol leírható ez :
 - > C++ esetén `Singleton::Instance()`
 - > Java esetén `Singleton.GetInstance()`
 - > C# esetén propertyvel célszerű: `Singleton.Instance`
 - → globális hozzáférés a példányhoz

Singleton C++

- C++ implementáció
- A Singleton konstruktora protected láthatóságú! Ez garantálja, hogy csak a statikus *Instance* metódushíváson keresztül lehessen példányt létrehozni.

```
class Singleton
{
public:
    static Singleton* Instance();
protected:
    Singleton() {};
private:
    static Singleton* _instance;
};

Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance()
{
    if (_instance == 0)
    {
        _instance = new Singleton;
    }
    return _instance;
}

int main(int argc, char* argv[])
{
    Singleton* s1 = Singleton::Instance();
}
```

Singleton C#

- C# implementáció
- A Singleton konstruktora `protected` láthatóságú! Ez garantálja, hogy csak a statikus `Instance` tulajdonságón keresztül lehessen példányt létrehozni.
- Vigyázat!
 - Ez nem szálbiztos!

```
public class Singleton
{
    private static Singleton instance = null;
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
                instance = new Singleton();
            return instance;
        }
    }

    protected Singleton() { }
    public void Print() {...}
}
```

Szálbiztos C# implementáció

- Nem kell tudni!
- Az előző megoldásunk nem volt szálbiztos!
- A statikus tagváltozók inicializálása az osztály első használatakor történik
- A C# fordító olyan kódot generál az inicializáláshoz, ehhez, ami szálbiztos.

```
public class Singleton
{
    private readonly static Singleton instance =
        new Singleton();

    public static Singleton Instance
    {
        get { return instance; }
    }

    protected Singleton() { }
    public void Print() {...}
}
```

TOVÁBBI MINTÁK

Observer

Adapter

Composite

Facade

Command

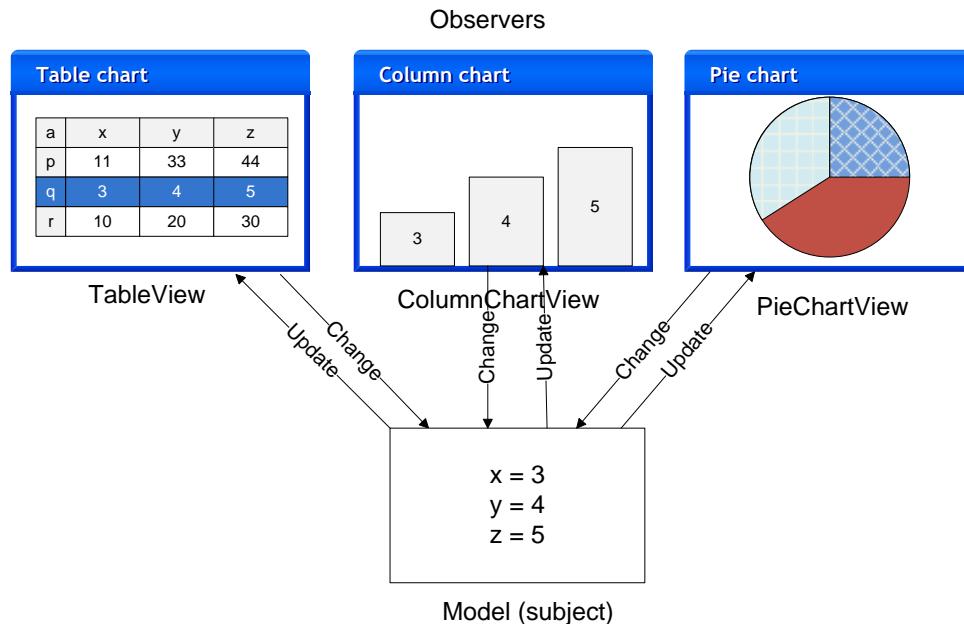
Memento

Stb.

Observer

Observer

- Cél
 - > Hogyan tudják objektumok értesíteni egymást állapotuk megváltozásáról anélkül, hogy függőség lenne a konkrét osztályaiktól
- Példa: MVC vagy Document-View architektúra
 - > A felhasználó megváltoztatja az egyik nézeten az adatokat, hogyan frissítsük a többet? Közvetlen függvényhívással?



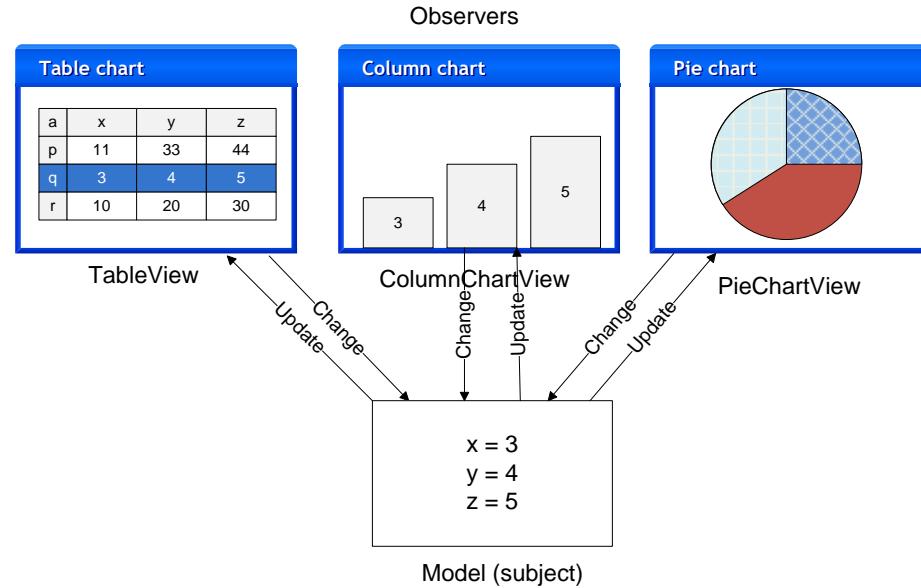
Observer

- Közvetlen függvényhívás hátrányok
 - Függőség a konkrét osztálytól. Pl. a *TableView* függ a *ColumnChartView* és a *PieChartView* osztályuktól
 - Ha új nézetet szeretnék bevezetni, minden nézet osztályt módosítani kell
 - A modell (üzleti logika) nem újrafelhasználható, mert össze van vonva a megjelenítéssel. Cél lenne, hogy úgy jelenjen meg, hogy ne legyen benne hivatkozás egy (konkrét) megjelenítési osztályra sem, mert akkor fel tudnánk több helyen használni
 - Nehéz karbantartani, továbbfejleszteni, újrafelhasználni, mert túl szoros a csatolás az osztályok között

Observer

- Megoldás

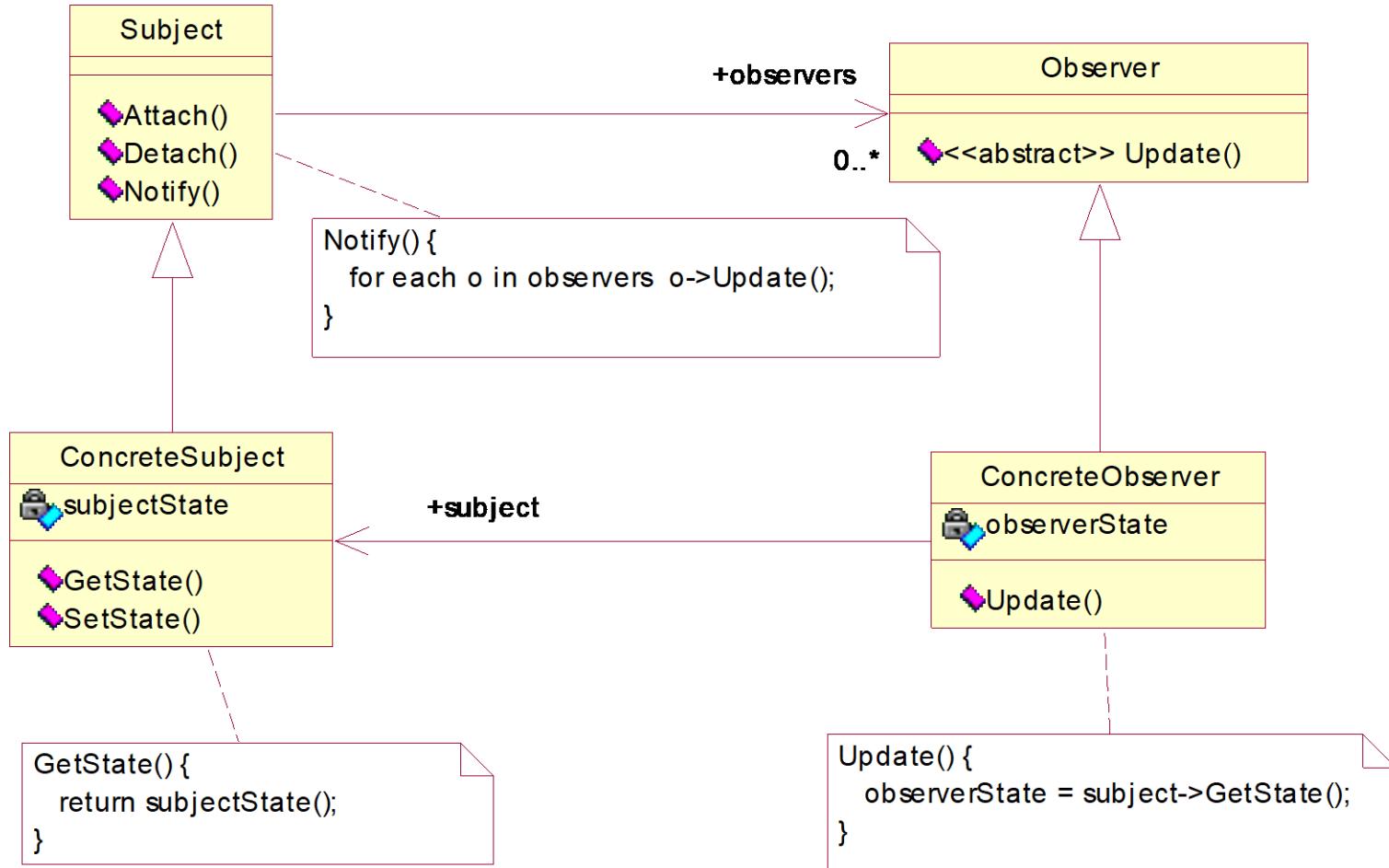
- > Az előző példára a MVC vagy Document-View architektúra
- > Emeljük ki az adatokat és az azon értelmezett műveleteket egy modell osztályba
- > A modellhez különböző view-kat (observers) lehet beregisztrálni
- > Ha valamelyik view megváltoztatja a modell adatait, a modell értesíti az összes beregisztrált view-t a változásról.
- > Az értesítés hatására a view lekérdezi a modell állapotát és frissíti magát
- > A modell csak egy közös view (observer) interfészen keresztül tárolja a beregisztrált view-kat



Observer

- Előnyök
 - > A modell kódjában csak egy *IView* lista van, így a modell független az egyes *IView-t* implementáló osztályoktól. A modell újrafelhasználható!
 - > Egy egyszerű mechanizmust kaptunk arra, hogy az összes view konzisztens nézetét mutassa az adatoknak.
 - > A rendszer könnyen kiterjeszthető új view osztályokkal. Sem a modellt, sem a többi view osztályt nem kell ehhez módosítani.
- Általánosítva, elvonatkoztatva a model-view esettől
 - > A fenti megközelítés lehetővé teszi, hogy egy objektum megváltozása esetén értesíteni tudjon tetszőleges más objektumokat anélkül, hogy bármit is tudna róluk
 - > Ez az ún. *observer* minta lényege

Observer – statikus nézet

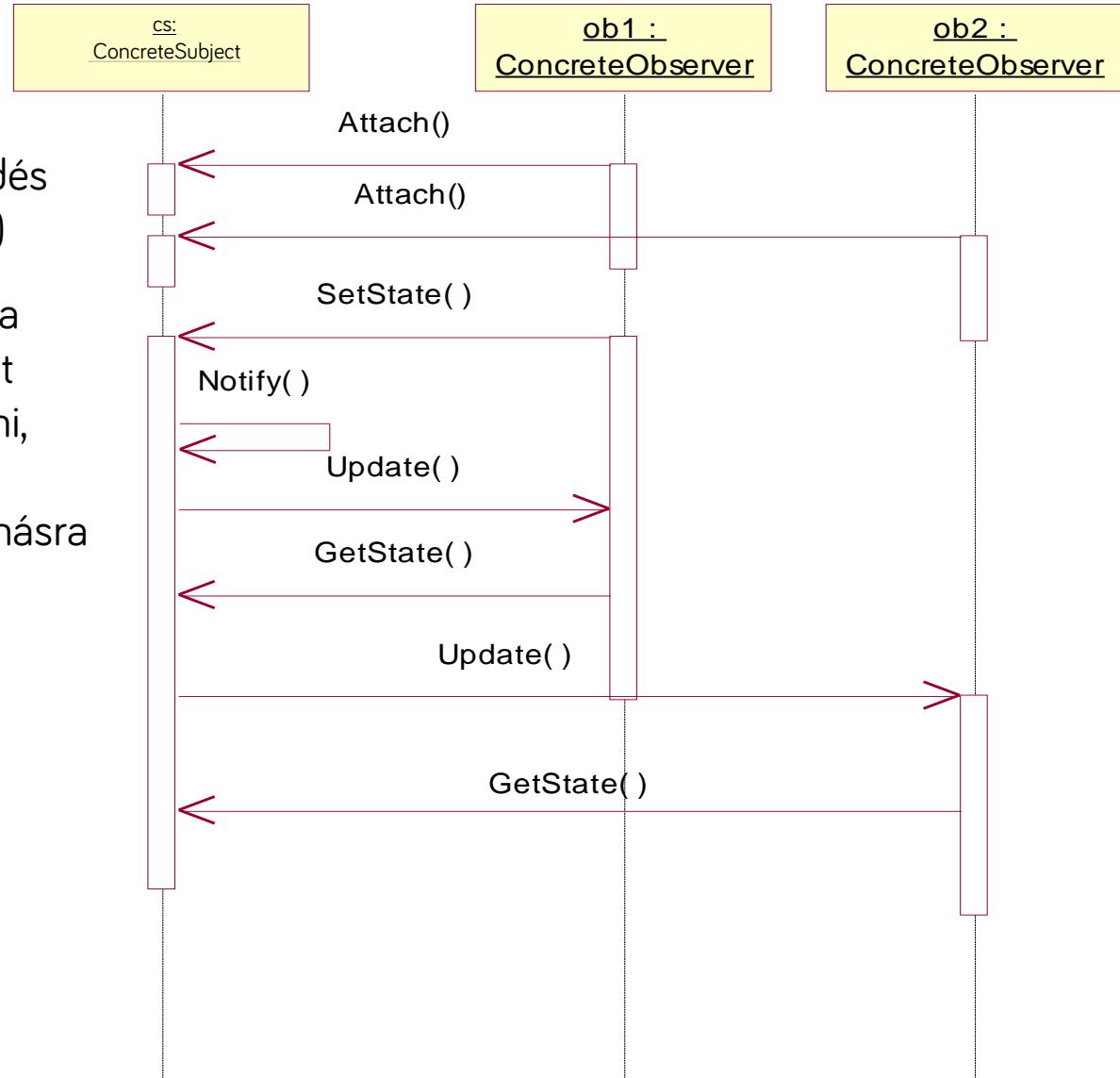


Observer - szereplők

- Subject
 - > Tárolja a beregisztrált Observer-eket
 - > Interfész definiál Observer-ek be- és kiregisztrálására valamint értesítésére
- Observer
 - > Interfész definiál azon objektumok számára, amelyek értesülni szeretnének a Subject-ben bekövetkezett változásról (Update művelet)
- ConcreteSubject
 - > Az observer-ek számára érdekes állapotot tárol
 - > Értesíti a beregisztrált observer-eket, amikor az állapota megváltozik
- ConcreteObserver
 - > Referenciát tárol a megfigyelt ConcreteSubject objektumra
 - > Olyan állapotot tárol, amit a megfigyelt ConcreteSubject állapotával konzisztensen kell tartani
 - > Implementálja az Observer interfészét (Update művelet), ez az, amit a Subject meghív, amikor a ConcreteSubject állapota megváltozik. Ebben frissíti a saját állapotát a megfigyelt ConcreteSubject állapotának megfelelően

Observer

- Dinamikus nézet – viselkedés (beregisztrálás és értesítés)
- „Sajnos” az UML szekvencia diagram csak objektumokat ábrázol, nem képes kifejezni, hogy milyen interfészen keresztül hivatkoznak egymásra az objektumok, így a függetlenséget nem tudja kifejezni



Observer

- Használjuk, ha
 - > Amikor egy objektum megváltoztatása maga után vonja más objektumok megváltoztatását, és nem tudjuk, hogy hány objektumról van szó
 - > Amikor egy objektumnak értesítenie kell más objektumokat az értesítendő objektum szerkezetére vonatkozó feltételezések nélkül
- Megjegyzés: az Observer az egyik leggyakrabban használt minta

Adapter

(wrapper)

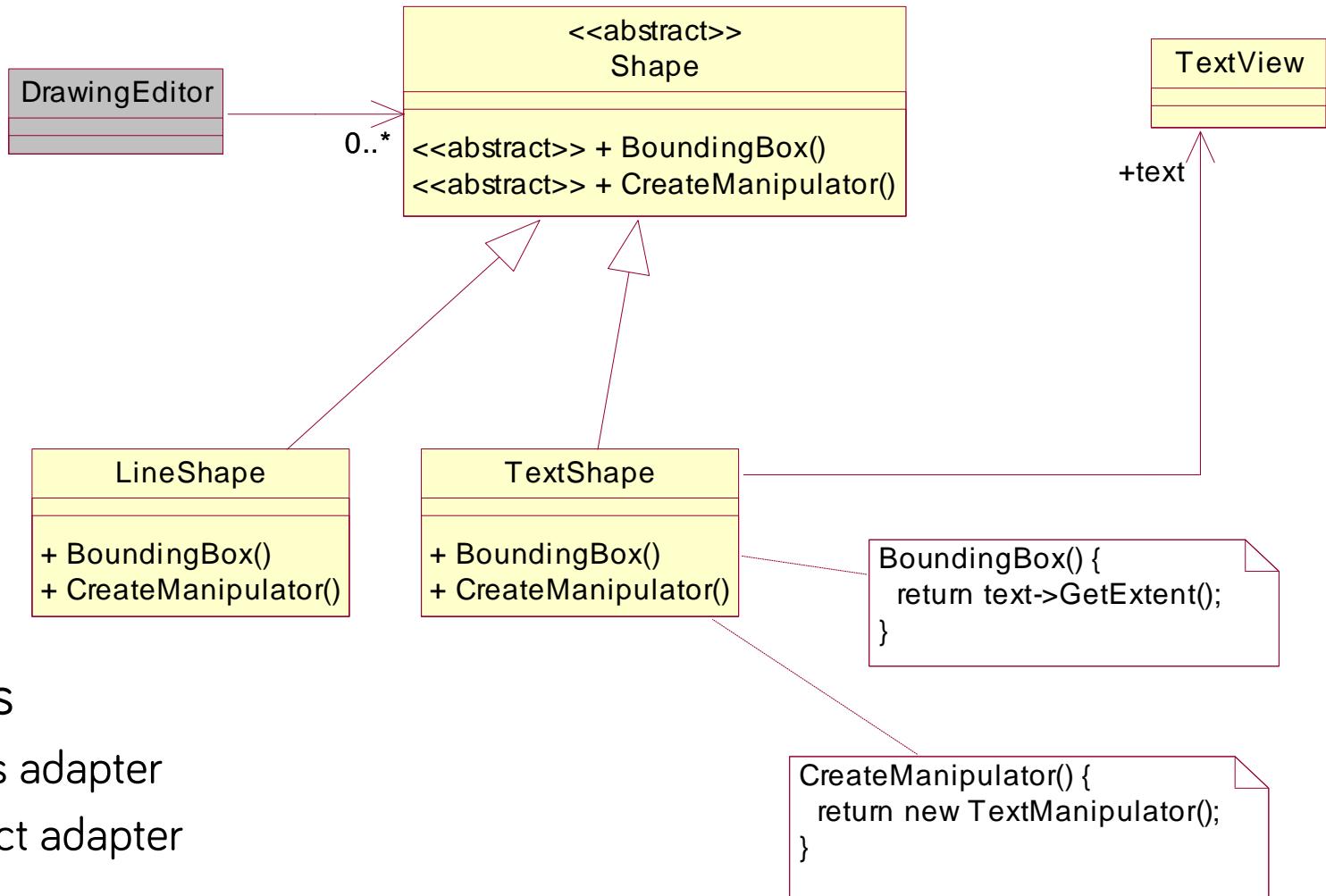
Adapter

- Cél
 - > Egy osztály interfészét olyan interfésszé konvertálja, amit a kliens vár. Lehetővé teszi olyan osztályok együttműködését, melyek egyébként az inkompatibilis interfészeik miatt nem tudnának együttműködni.

Adapter - példa

- Grafikus Editor
 - > Grafikus alakzatok (vonalak, poligon, szöveg) – a Shape osztályból származnak (vonal – LineShape, poligon – PoligonShape, szöveg – TextShape, stb)
 - > TextShape megírása nehéz, viszont tegyük fel, hogy a framework egy sokoldalú TextView osztály, ami mindenre tudja, amit a TextShape-től elvárunk
 - > Probléma: a TextView osztályt nem tudjuk közvetlenül felhasználni, mert nem megfelelő az interfésze, ugyanis nem a Shape osztályból származik (nem támogatja a Shape interfészt, emiatt nem tudjuk a többi Shape-el együtt egységesen kezelní)
 - > Megoldás: Adapter minta használata

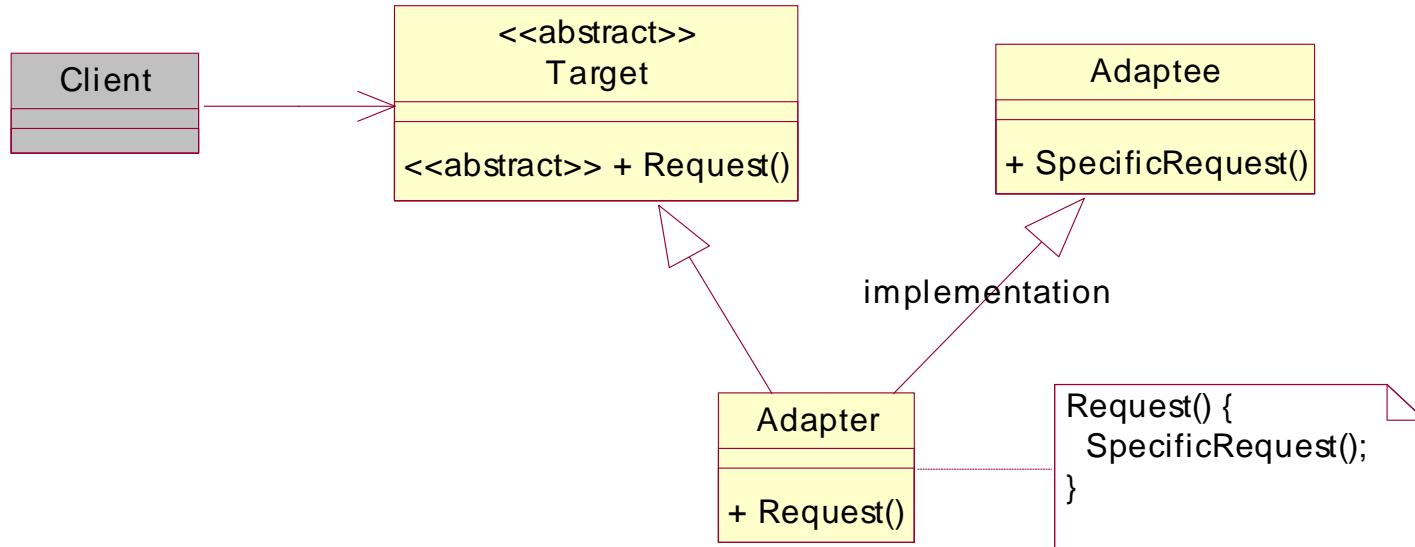
Adapter



- Két típus
 - > Class adapter
 - > Object adapter
- A fenti példa egy „object” adapter

Adapter

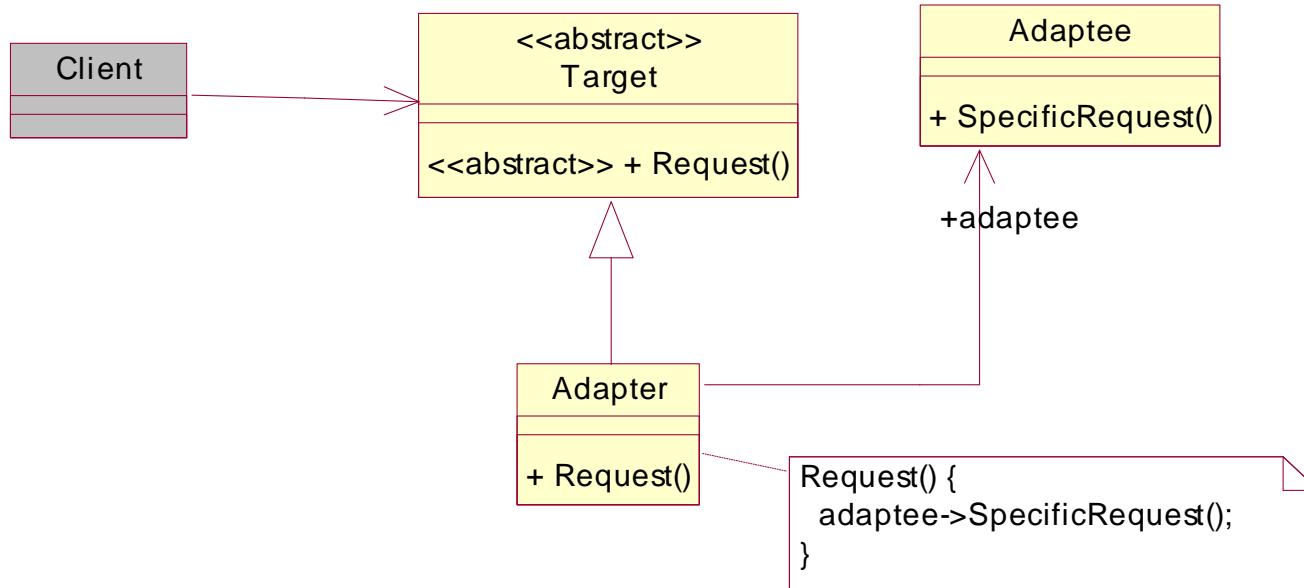
- Struktúra – class adapter (első verzió)



- Többszörös örökléssel oldja meg az adaptálást
- Szereplők
 - > **Adaptee** (*TextView*): interfész, amit adaptálni (illeszteni) kell
 - > **Adapter** (*TextShape*) : illesztő, az Adaptee interfészt a Target interfésszé konvertálja
 - > **Target** (*Shape*): interfész, amit a kliens használ

Adapter

- Struktúra – object adapter (második verzió)



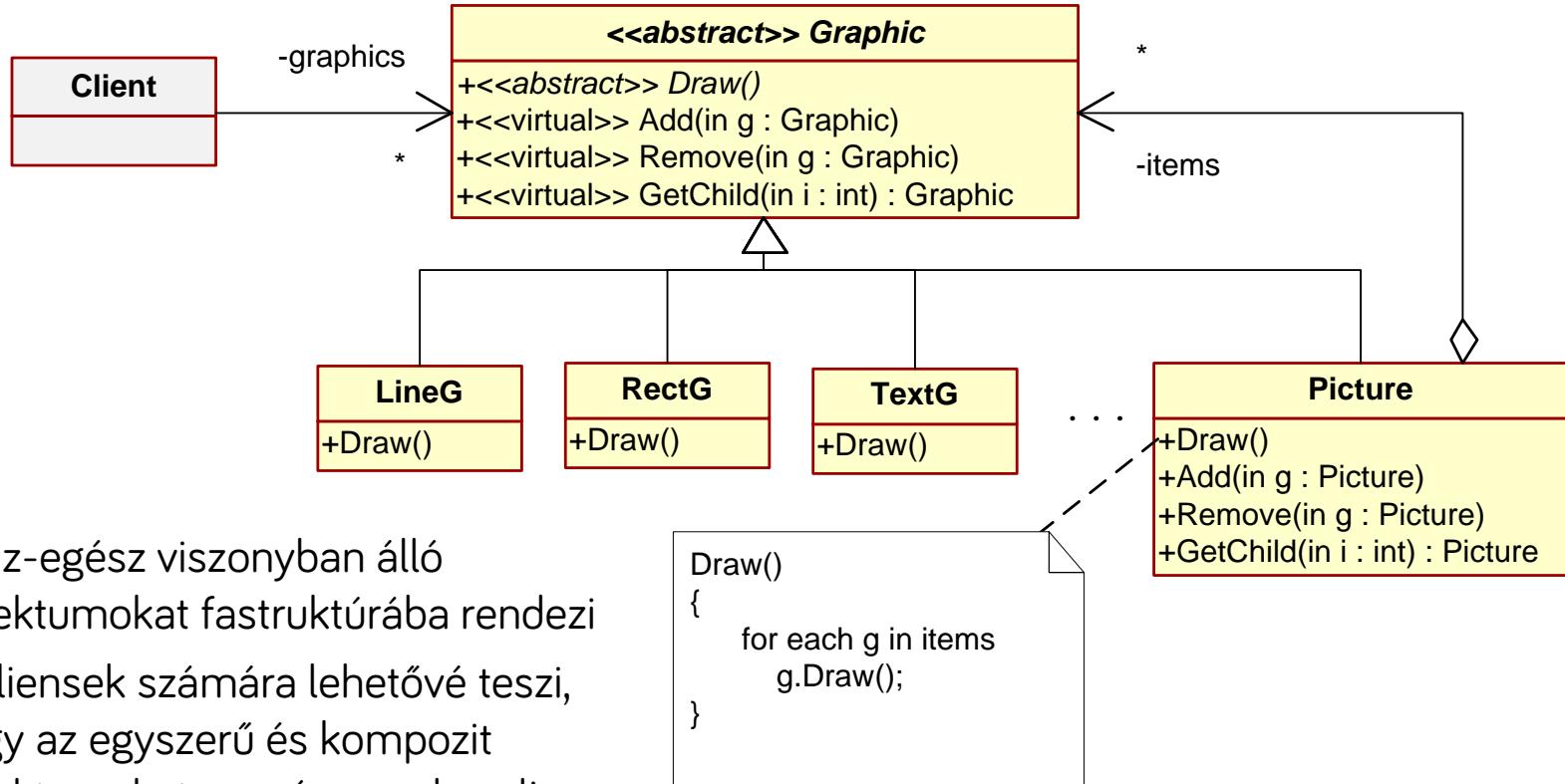
- Objektum kompozícióval, delegálással oldja meg az adaptálást
- Szereplők: mint a *Class Adapter*-nél
 - > Az Adapter tartalmaz egy pointert vagy referenciát az Adaptee-ra
 - > Az adapter delegálja a művelet végrehajtását az Adaptee-ra
 - > Egy adapter képes több Adaptee-t is magában foglalni, beleértve azok alosztályait is

Adapter

- Implementáció
 - > C++-ban a class adapter esetén
 - Private öröklés az Adaptee-ból (implementáció öröklés)
 - Public öröklés a Target-ból (interfész öröklés)

Composite

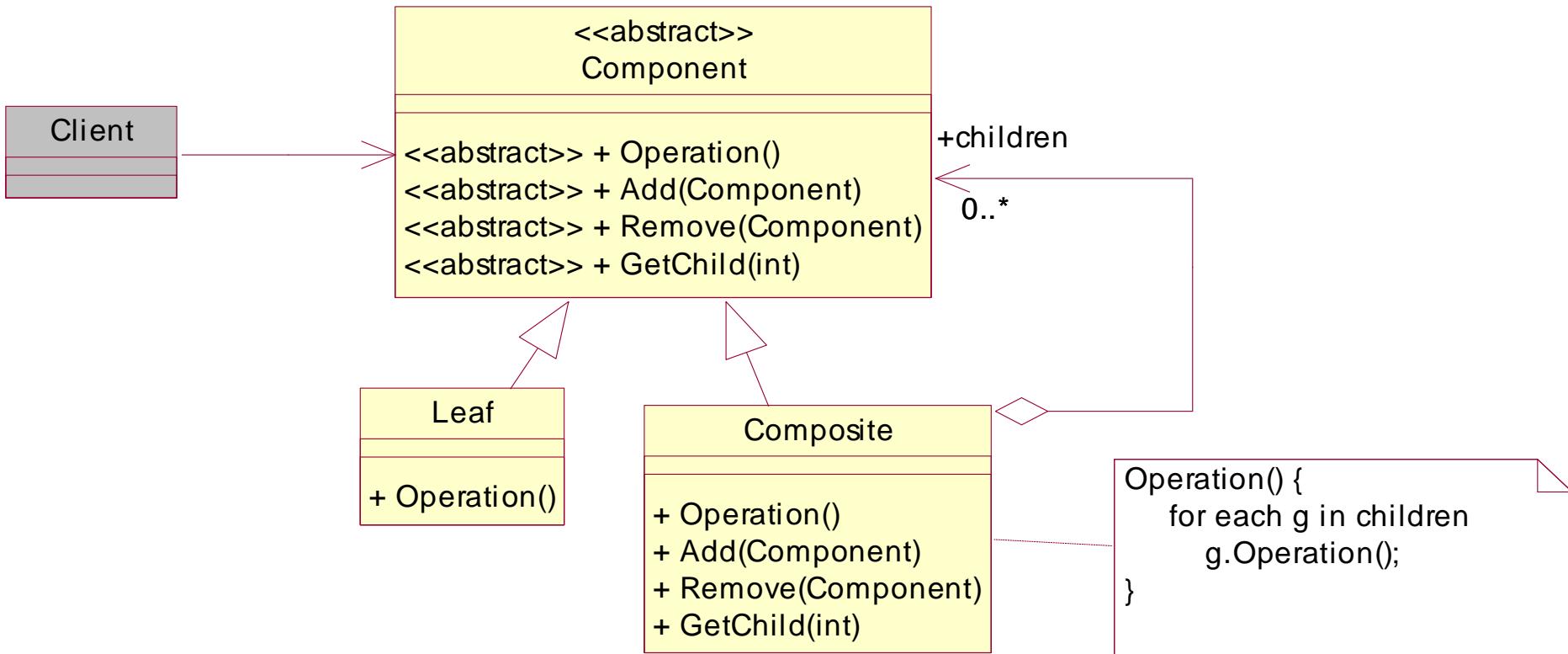
Composite



- Célja
 - > Rész-egész viszonyban álló objektumokat fastruktúrába rendezi
 - > A kliensek számára lehetővé teszi, hogy az egyszerű és kompozit objektumokat egységesen kezelje
- Példa
 - > Olyan grafikus alkalmazás, amely lehetővé teszi összetett grafikus objektumok létrehozását

Composite

- Struktúra



Composite

- Használjuk, ha
 - > Objektumok rész-egész viszonyát szeretnénk kezelni
 - > A kliensek számára el akarjuk rejteni, hogy egy objektum egyedi objektum vagy kompozit objektum: bizonyos szempontból egységesen szeretnénk kezelni őket.

Composite

- **Megjegyzés:** Kérdés, hogy melyik osztályban definiáljuk a kompozit kezelő műveleteket (Add, Remove, ...), amelyek nem értelmezhetők a levél objektumokra (LineG, TextG, stb)
 - > A “**Component** osztályban (ez volt a példában):
 - előny, hogy egységesen kezelhető minden objektum (kompozit és nem kompozit objektumok), hiszen mind támogatja az Add, Remove, GetChild műveleteket
 - hátránya, hogy nem biztonságos, de megoldás lehet pl. Exception dobása, ha a művelet nem értelmezhető
 - > A **Composite** osztályban:
 - Hátrány: fordítva, mint az előbb
 - Pl. megoldás arra, hogy a kliens el tudja dönteni, hogy egy objektum kompozit-e: **IsComposite**: bool virtuális függvény definiálása a **Component** osztályban, a levél osztályokban false-al térjen vissza.

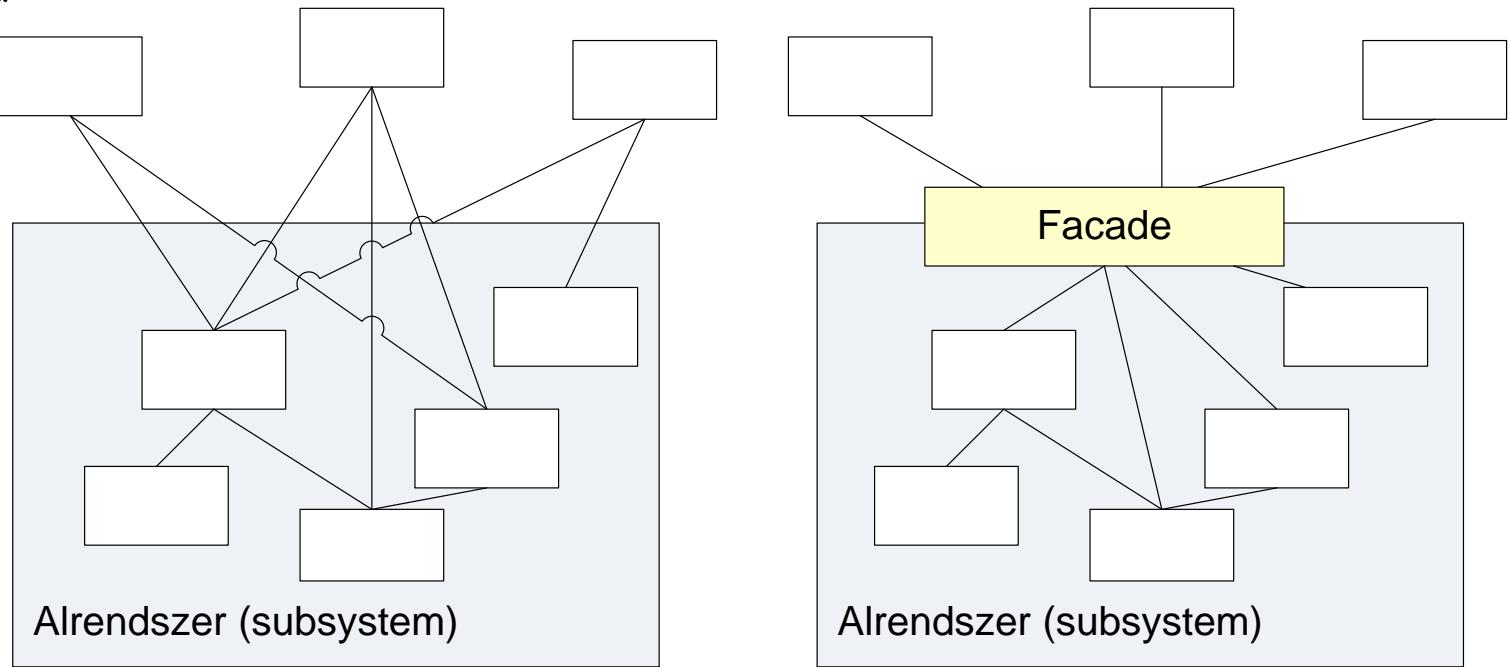
Composite

- Kérdés
 - > Hol jelenik meg a Composite minta a Windows Forms alkalmazásokban?
- Feladat1
 - > Egy 3D-s CAD program esetén összetett, sokkomponensű térbeli testek térfogatát kell kiszámítani. Milyen módon segít a megoldásban a Composite minta?

Facade

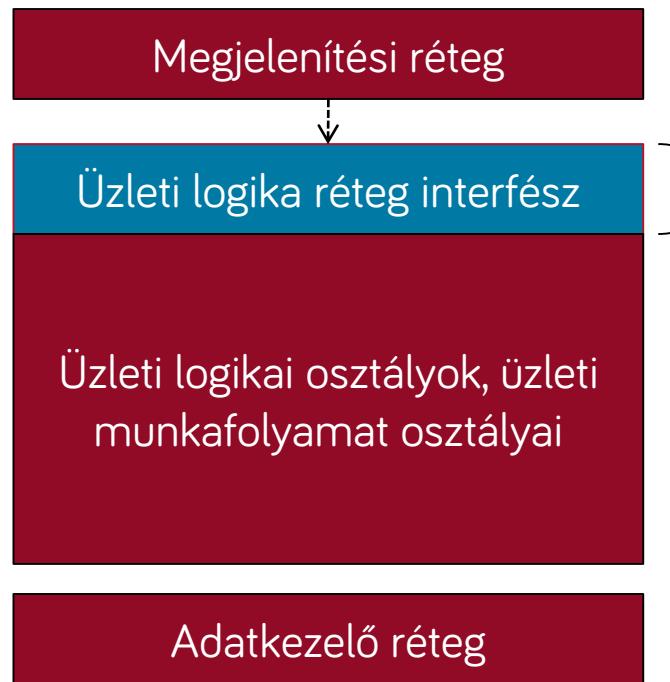
Facade

- Célja
 - > Egységes interfészt definiál egy alrendszer interfészeinek halmazához
 - > Magasabb szintű interfészt definiál, amin keresztül könnyebb az alrendszer használata
- Példa



Facade

- Példa 1
 - > Compiler
 - Command line hívható cpp.exe, nagyon sok argumentummal
 - A belsejéhez nem férünk hozzá (parser, tokenizer, stb.)
- Példa 2
 - > Többrétegű (többnyire üzleti) alkalmazások



A megjelenítsi réteg csak ennek a vékony rétegnak az osztályait látja

Facade

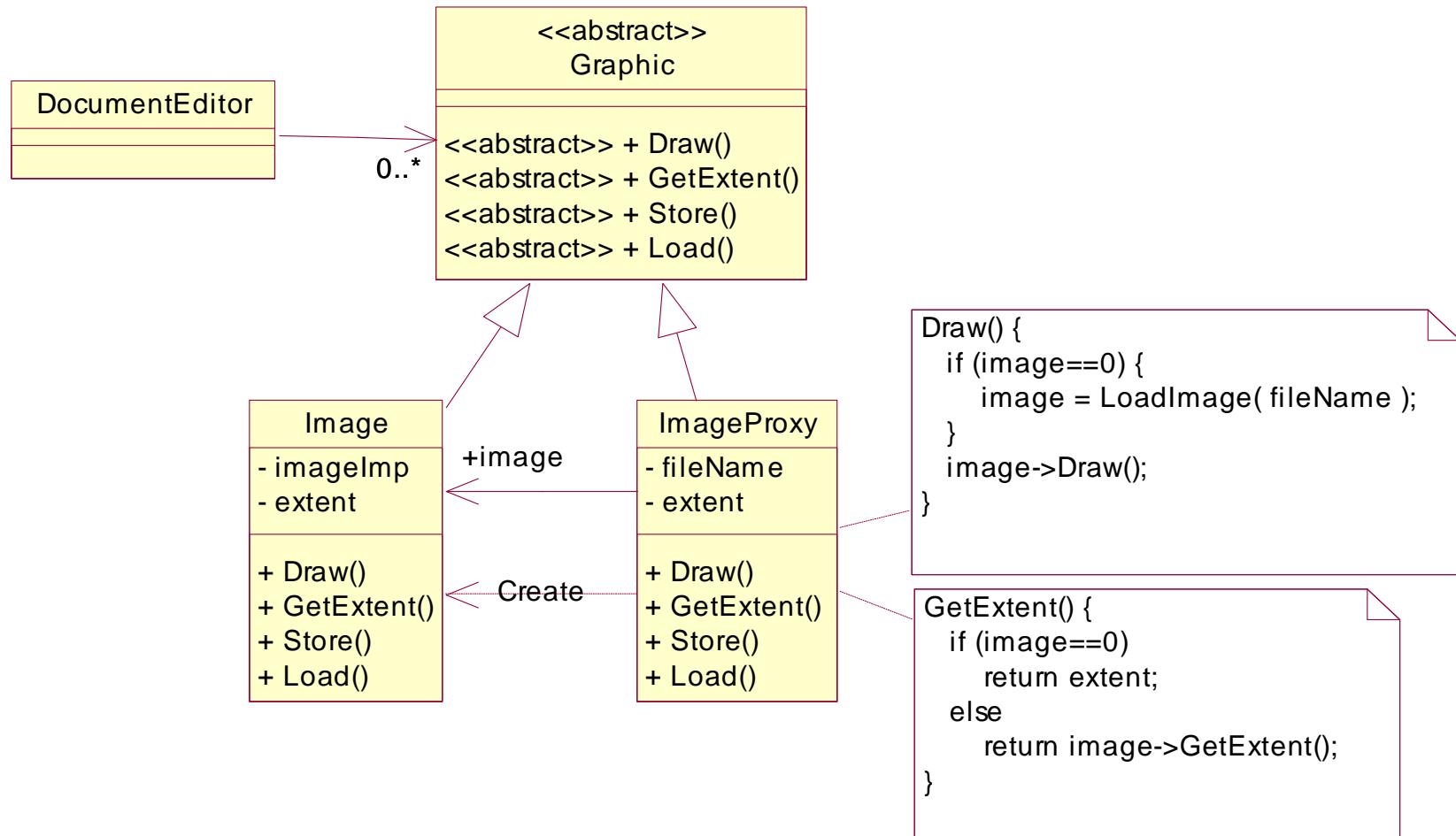
- Használjuk, ha
 - > Egyszerű interfészt szeretnénk biztosítani egy komplex rendszer felé
 - > Számos függőség van a kliens és az alrendszerek osztályai között. Ilyenkor ha létrehozva egy Facade-ot elősegítve az alrendszer függetlenségét és a hordozhatóságot
 - > Layers (rétegelés) esetén
- Megjegyzés
 - > Külön döntés, hogy engedünk-e hozzáférést az alrendszerek osztályaihoz
 - Elvileg nem akadályozza meg, hogy a kliensek felhasználják az alrendszerek osztályait, ha arra is szükségük van

Proxy

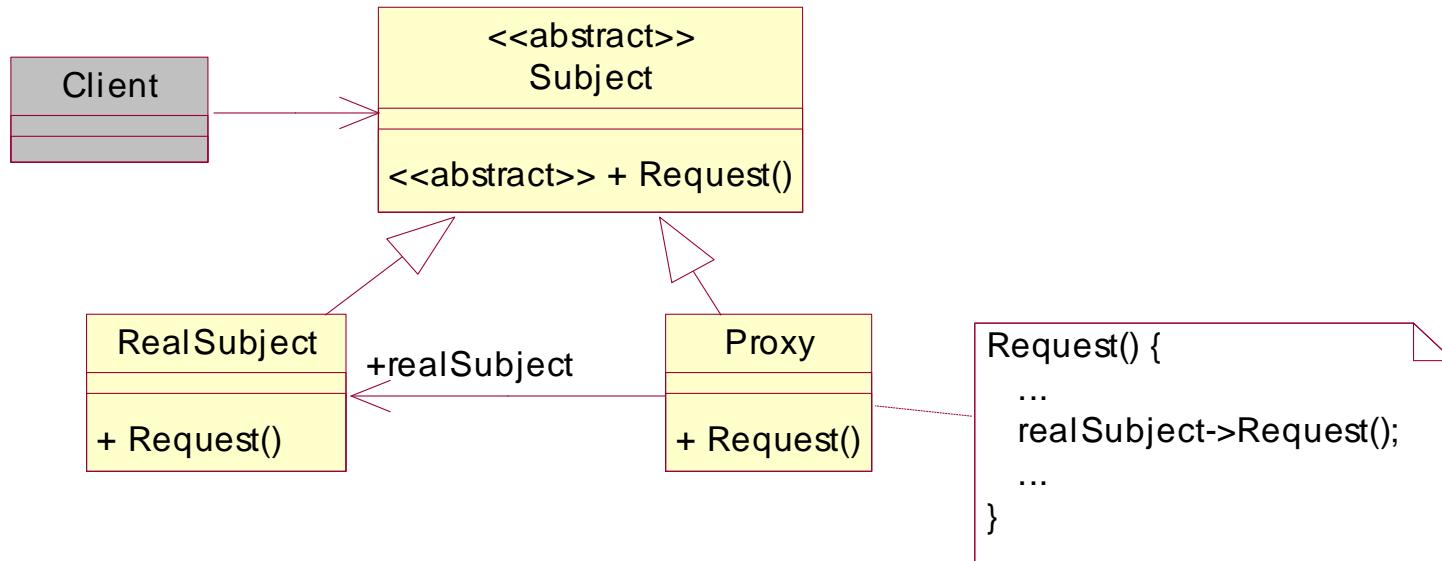
Proxy

- Célja
 - > Objektum helyett egy helyettesítő objektumot használ, ami szabályozza az objektumhoz való hozzáférést
- Példa
 - > Szövegszerkesztő
 - Sok nagy méretű kép
 - Nem kel őket egyszerre megjeleníteni, csak amikor odagörgetjük az ablakot, vagyis amikor láthatóvá válik
 - > Megoldás: Proxy
 - Helyettesítsük a képet egy objektummal (proxy), amely ha be van töltve a kép megjeleníti, egyébként betölti, és azután jeleníti meg

Proxy



Proxy - struktúra



- **Subject:** közös interfészt biztosít a RealSubject és a Proxy számára (ezáltal tud a minta működni, ez a lényeg!)
- **Realsubject:** a valódi objektum, amit a proxy elrejt
- **Proxy:** helyettesítő objektum. Tartalmaz egy referenciát a tényleges objektumra, hogy el tudja azt érni. Szabályozza a hozzáférést a tényleges objektumhoz, feladata lehet a tényleges objektum létrehozása és törlése is.

Proxy

- Távoli Proxy
 - > Távoli objektumok lokális megjelenítése átlátszó módon. A kliens nem is érzékeli, hogy a tényleges objektum egy másik címtartományban, vagy egy másik gépen van
- Virtuális Proxy
 - > Nagy erőforrás igényű objektumok igény szerinti létrehozása (pl. kép)
- Védelmi Proxy
 - > A hozzáférést szabályozza különböző jogok esetén
- Smart Pointer
 - > Egy pointer egységbezárása, hogy bizonyos esetekben automatikus műveleteket hajtson végre (pl.:lockolás)

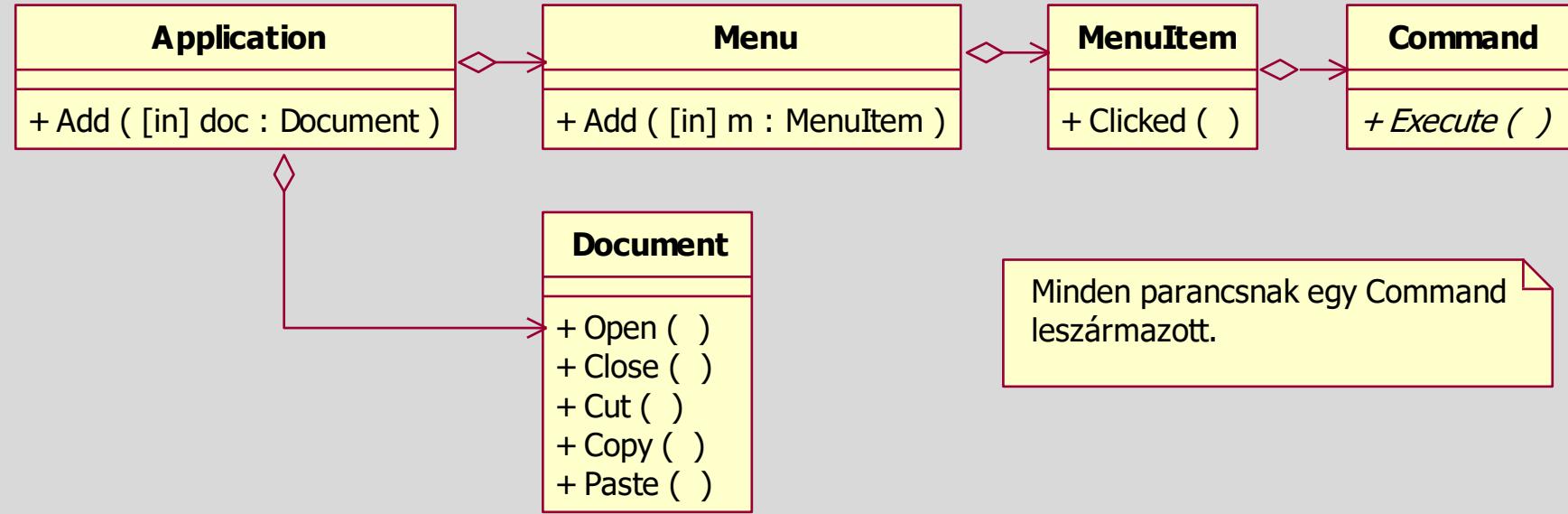
Command

Command

- Cél
 - > Egy kérés objektumként való egységbezárása. Ez lehetővé teszi a kliens különböző kérésekkel való felparaméterezését, a kérések sorba állítását, naplózását és visszavonását (undo)
- Alternatív nevek
 - > Action
- Nagyon rendszerfüggő (C++, .NET, stb.) a koncepció és az implementáció is
- Példa: felhasználói parancsok
 - > Menü, gomb, toolbar gomb
 - > Résztvevők pl.: Alkalmazás, Dokumentum, Menü, Almenü, Menüpont, stb
 - > Probléma: a GUI keretrendszer írói nem építhették bele az alkalmazásfüggő menüelem kiválasztás kezelést →
 - > Hogyan reagálunk a menüpont kiválasztása által generált eseményekre?
 - Callback függvény – nem objektum-orientált (strukturált) megoldás
 - Adapter alapú megoldások – Java
 - Delegate alapú megoldás - .NET
 - Command minta

Command

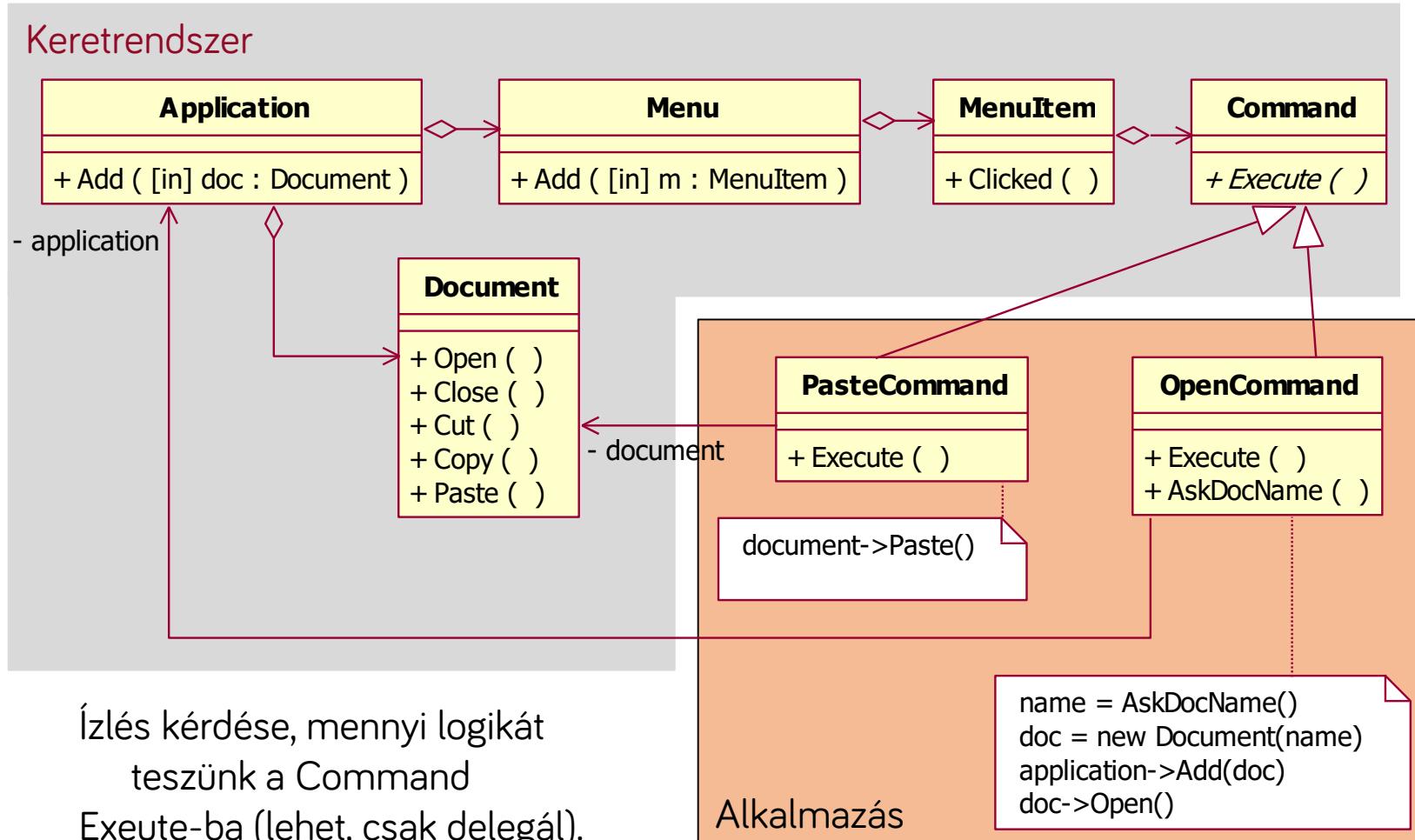
Keretrendszer



- Zárjuk külön **Command** leszármazott objektumba a kéréseket, és a menüelemeket ezzel paraméterezzük fel!
- **Feladat (kitérő)**
 - > Tervezzük át a fenti osztályhierarchiát úgy, hogy alkalmas legyen almenü kezelésére!

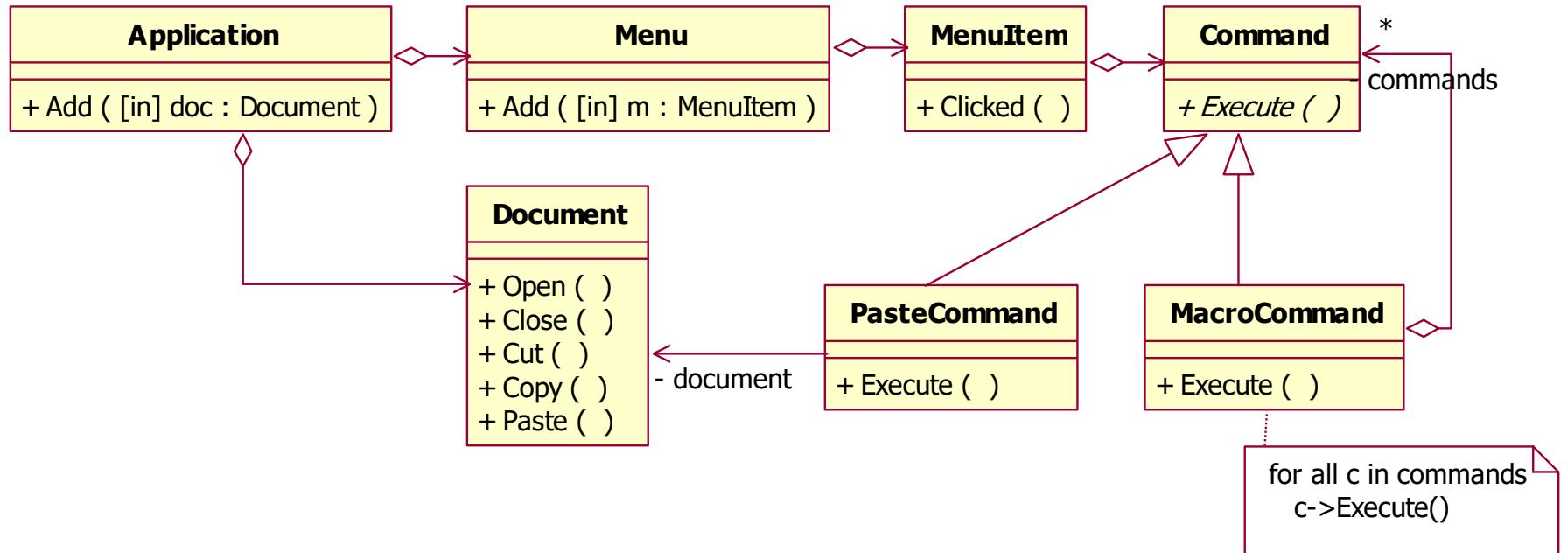
Command

- Egy konkrét megvalósítás: Paste, Open



Command

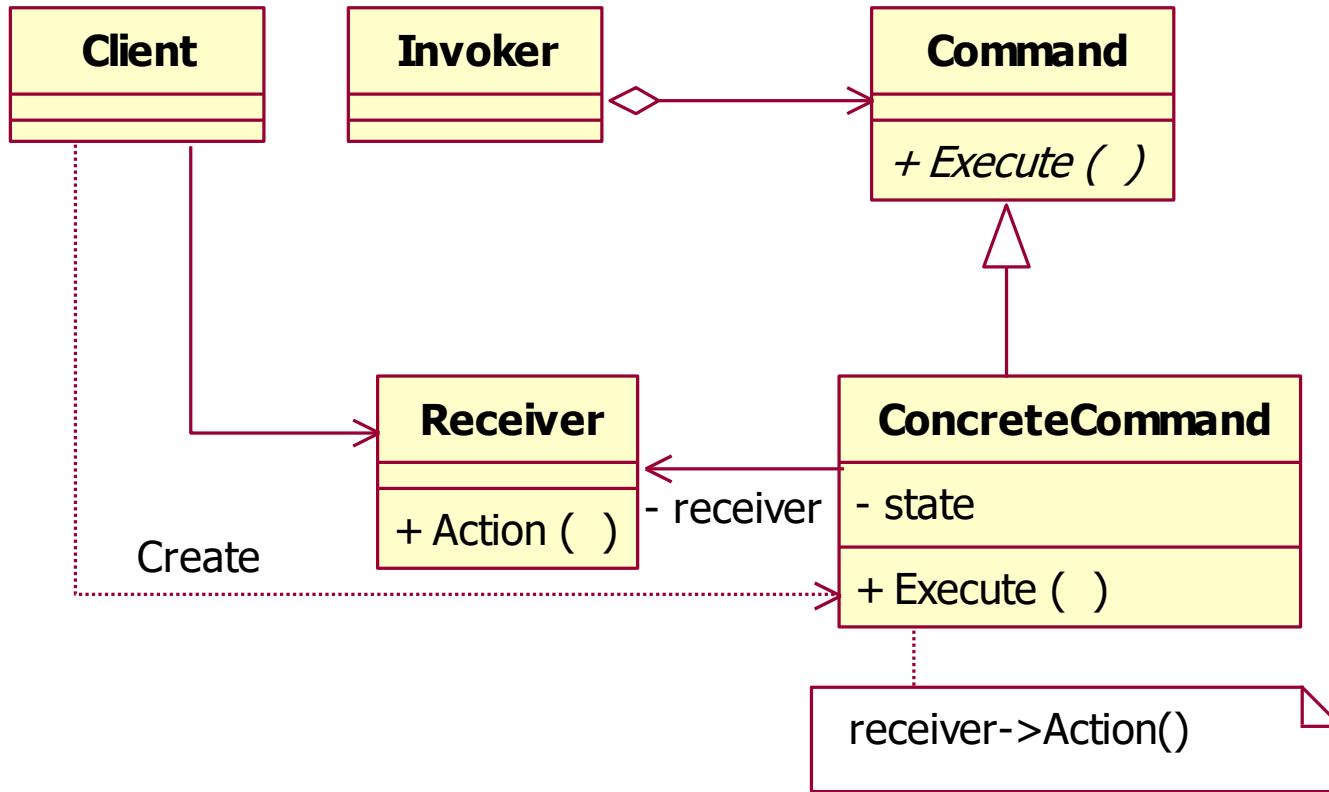
- Macro: parancsok szekvenciája



Command

- Használjuk, ha
 - > Ha strukturált programban callback függvényt használnánk, OO programban használunk commandot helyette.
 - > Szeretnénk a kéréseket különböző időben kiszolgálni. Ilyenkor várakozási sort használunk, a command-ban tároljuk a paramétereket, majd akár különböző folyamatokból/szálakból is feldolgozhatjuk őket.
 - Specifikus eset adott szálból szeretnénk futtatni. Pl. .NET-ben GUI szálból, bár itt a Control.Invoke a megfelelő megoldás.
 - > Visszavonás támogatására – eltároljuk az előző állapotot a command-ban

Command

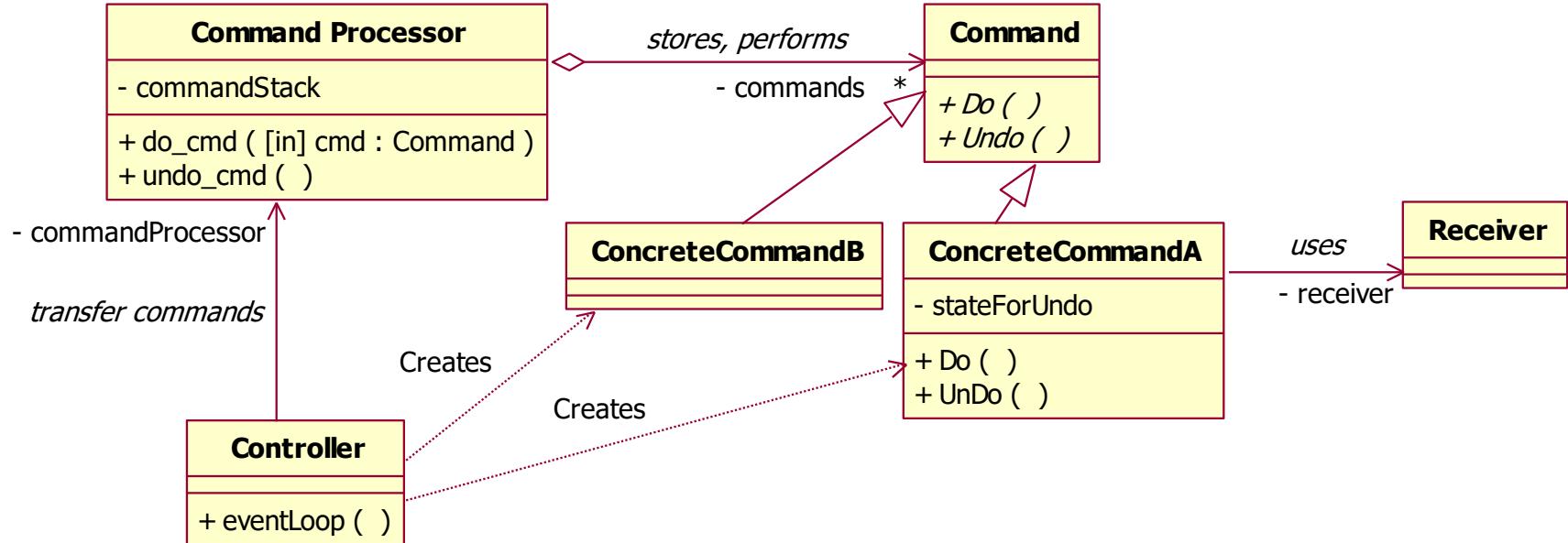


Command

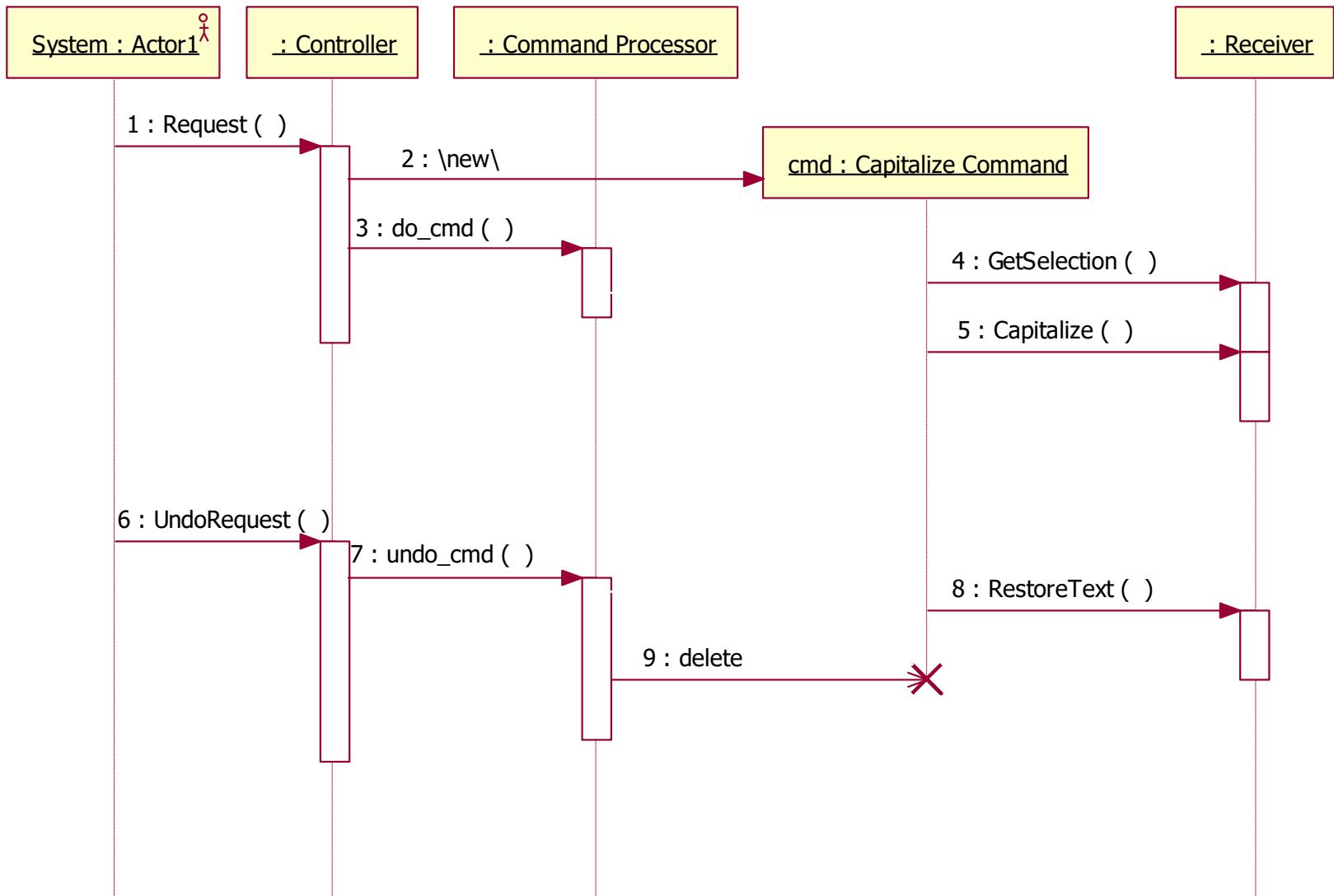
- Elválasztja a parancsot kiadó objektumot attól, amelyik tudja, hogyan kell kezelni
- Kiterjeszthetővé teszi a Command specializálásával a parancs kezelését
- Összetett parancsok támogatása
- Egy parancs több GUI elemhez is hozzárendelhető: tipikusan menüelem és toolbar gomb
- Könnyű hozzáadni új parancsokat, mert ehhez egyetlen létező osztályt sem kell változtatni. Hogyan tegyük ezt meg?

Command

- Command Processor
 - > A Command egy változata
 - > „Beépítve” támogatja az undo alapjait
 - > Kulcsa a Command Processor osztály
 - Nyilvántartja a Command objektumokat
 - Aktiválja a Command objektumokat és egyéb szolgáltatásokat nyújt



Command Processor



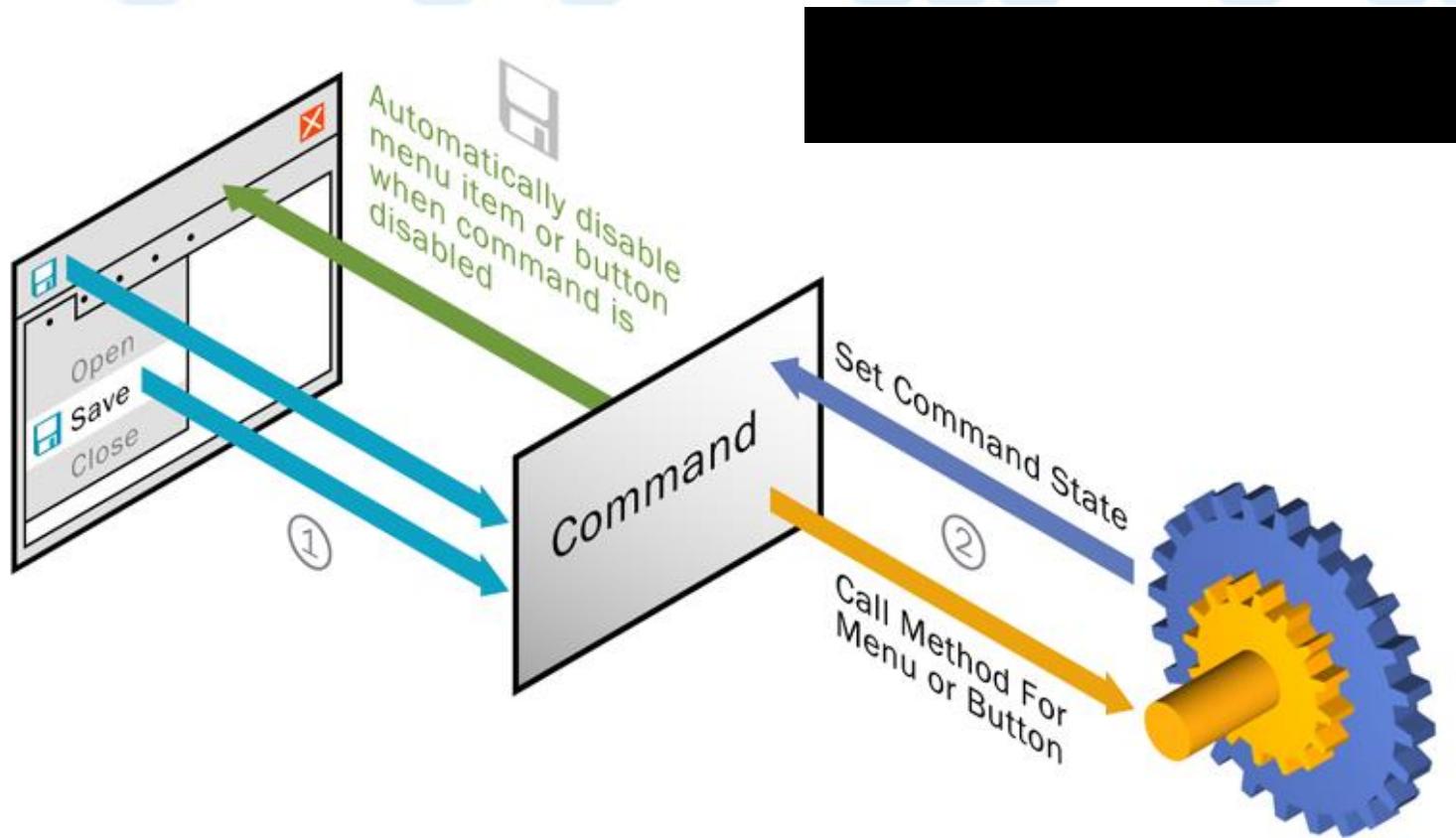
CommandBinding .NET-ben

- A CommandBinding teljesen más célokat szolgál, mint a Command vagy Command Processor !
- Ugyanazt a funkciót kiváltó menü, toolbar gomb, legördülő menü elemek nincsenek automatikusan összekötve
 - > Példa: „Save”
- Miért gond ez?
 - > Közös eseménykezelőre volna szükség: nem lehetséges, mert más az eseménykezelő szignatúra

```
private void MenuItemSave_Click(object sender, EventArgs e) { ... }  
voidToolBar1_ButtonClick(object sender, ToolBarButtonEventArgs e) {...}
```

- > A vezérlőelemek tiltása/engedélyezése nincs központosítva
- > Alkalmazzuk a CommandBinding mintát
- > Bővebben a labor anyagban

CommandBinding .NET-ben



Memento

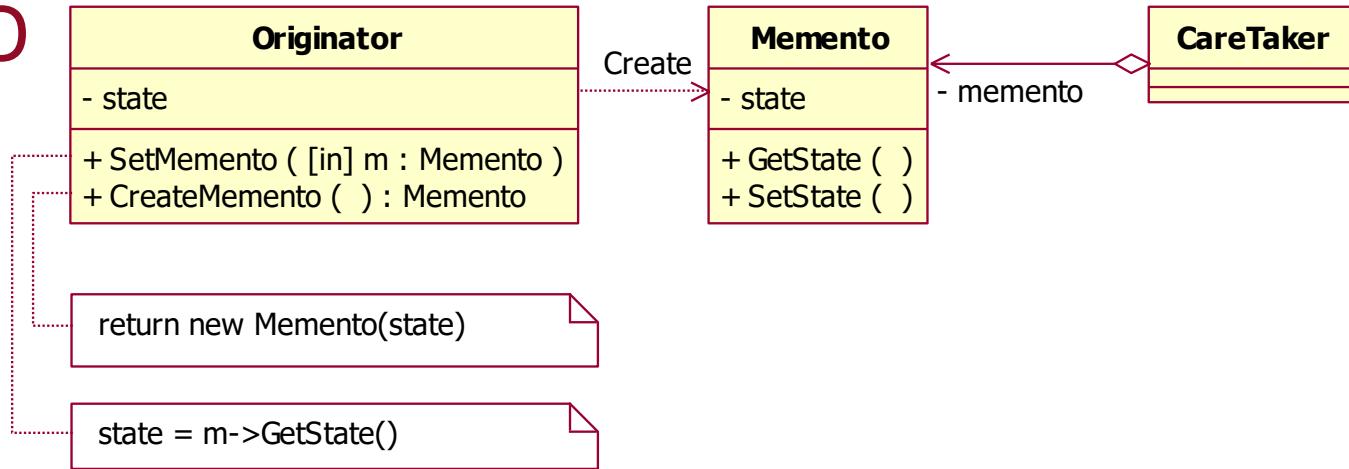
Memento

- Cél
 - > Az egységbezárás megsértése nélkül a külvilág számára elérhetővé tenni az objektum belső állapotát. Így az objektum állapota később visszaállítható.
- Példa: Visszavonás (undo) funkció a Dokumentumban

Memento

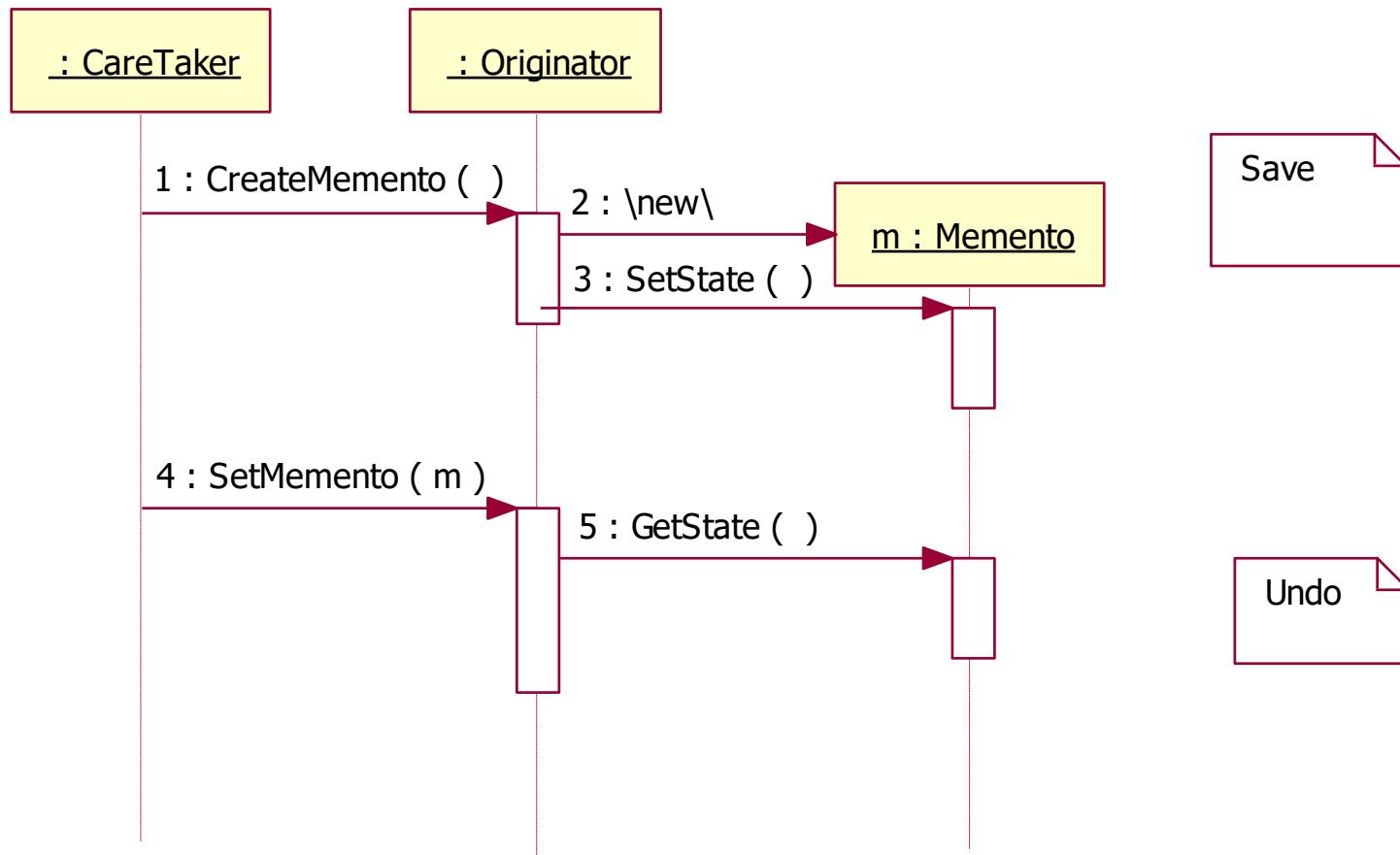
- Invertálható műveletek a Command mintában
 - > A visszavonás sokszor nehéz vagy lehetetlen anélkül, hogy az objektum (pl. dokumentum) **teljes állapotát** elmentenék, majd visszaállítanánk az Undo során (pl. dokumentum Clear parancs visszavonása).
 - > Az objektum teljes állapota viszont általában nem elérhető, mert az egységbezárás miatt a tagváltozók védettek (private).
 - > Csak az Undo-hoz való állapotmentés lehetősége miatt kellene ezeket a változókat publikussá tenni. Nem tesszük, inkább alkalmazzuk a Memento mintát.
- A Memento minta lényege, hogy egy **objektum** (pl. **dokumentum**) adott állapotát egy **Memento** objektumba csomagoljuk be, és ilyen „becsomagolt” formában tesszük elérhetővé (az Undo megvalósításához)

Memento



- **Originator:** az ō állapotát kell tudni visszaállítani.
 - > A `CreateMemento()` elment
(pontosabban visszaadja a state állapotot egy Memento objektum formájában)
 - > A `SetMemento()` visszaállít
(pontosabban beállítja a state állapotot a paraméterben megkapott Memento objektum alapján)
- **Memento:** az Originator állapotát tárolja és elméletileg csak az Originator számára biztosít hozzáférést az állapothoz (`state`). Pl. C++-ban a friend alkalmazásával oldható ez meg.
- **CareTaker:** nyilvántartja a Mementokat

Memento



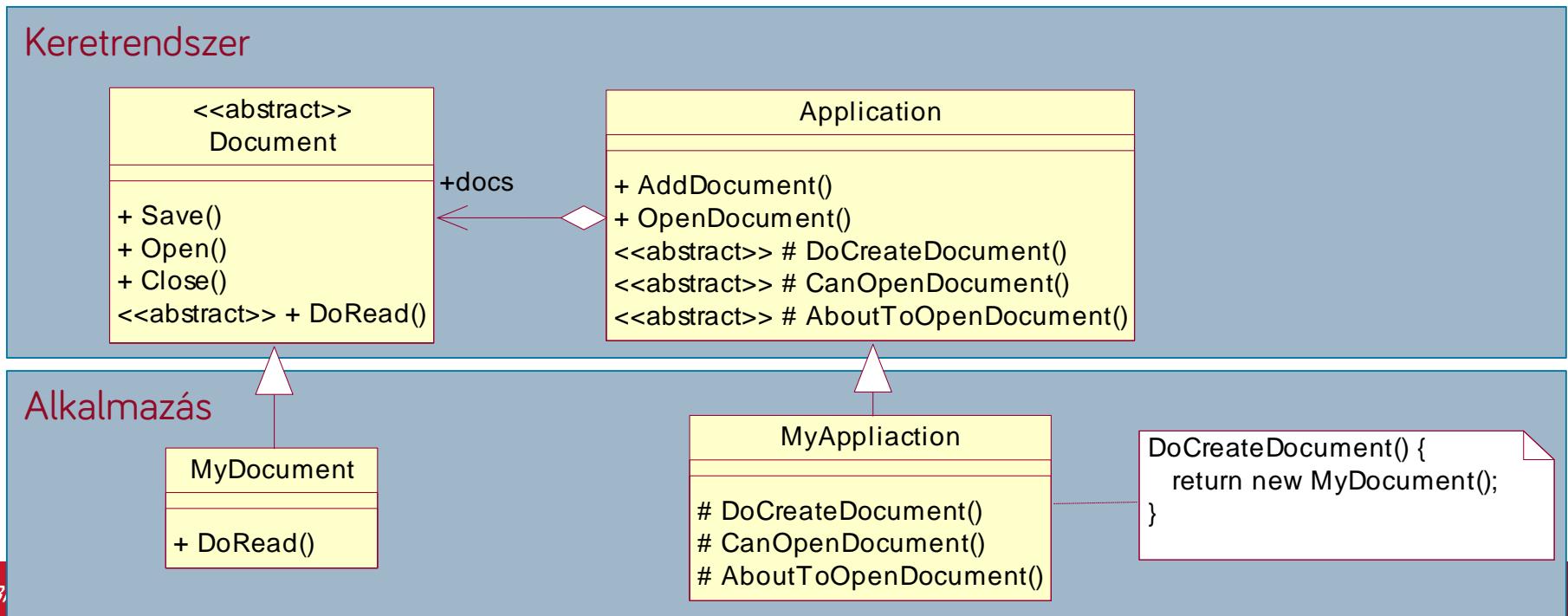
Memento

- Használjuk, ha
 - > Egy objektum (rész)állapotát később vissza kell állítani és egy közvetlen interfész az objektum állapotához használná az implementációs részleteket, vagyis megsértené az objektum egységbezárását
- Előnyök:
 - > Megőrzi az egységbezárás határait
 - > Egyszerűsíti az Originatort
- Hátrányok:
 - > Memento használata sokszor erőforrásigényes
 - > Nem minden jósolható meg a Caretaker által lefoglalt hely, és ez sok is lehet

Template method

Template method

- Cél
 - > Egy műveleten belül algoritmus vázat definiál, és ennek néhány lépéseinak implementálását a leszármazott osztályra bízza.
- Példa: Framework-ben dokumentum megnyitása
 - > A framework-ben legyen adott két osztály, **Application** és **Document**. Ezekből kell a programozónak egy-egy saját osztály leszármaztatnia, amikben megvalósítja az alkalmazásspecifikus viselkedést.



Template method

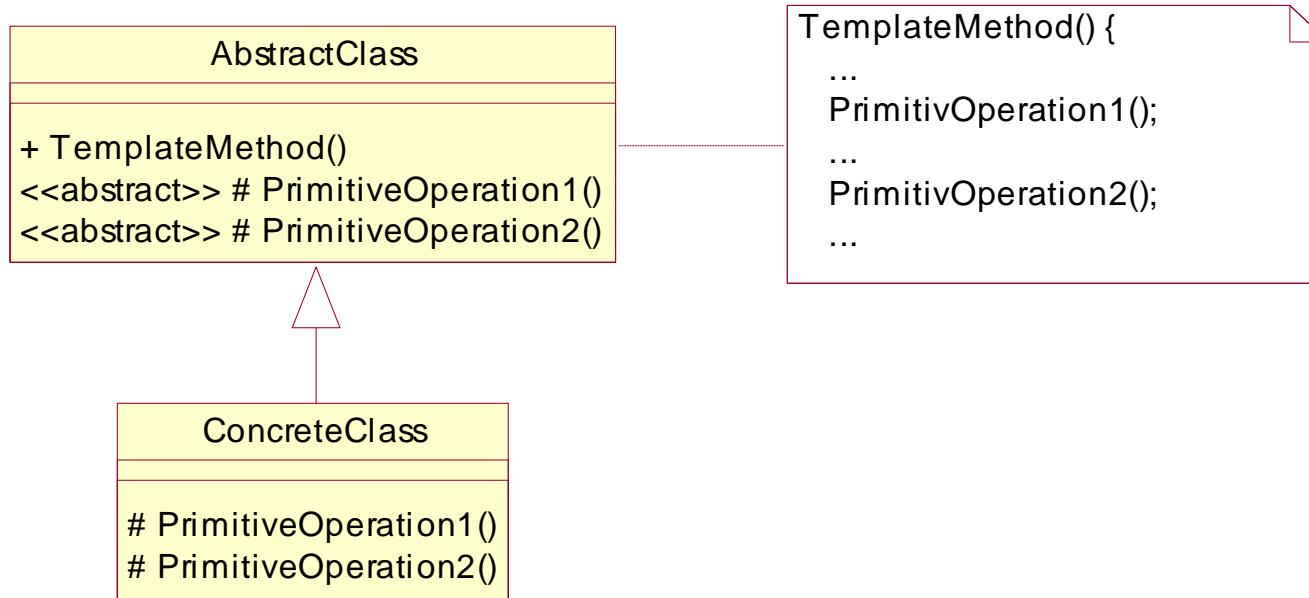
- A példában az OpenDocument egy ún. template method
 - > Meghatározza a műveletek sorrendjét
 - > Meghív néhány absztrakt műveletet, melyeket a leszármazott osztályban felül kell definiálni, hogy meghatározott viselkedést rendeljünk hozzá az aktuális igényeknek megfelelően

```
// Az Application osztály a framework része
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        // cannot handle this document
        return;
    }
    // a DoCreateDocument-et a adott alkalmazás megírásakor az
    // az Application-ból leszármazott MyApplication osztályban
    // felül kell definiálni, itt lesz majd értelmesen „kitöltve”
    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```

Template method

- Struktúra

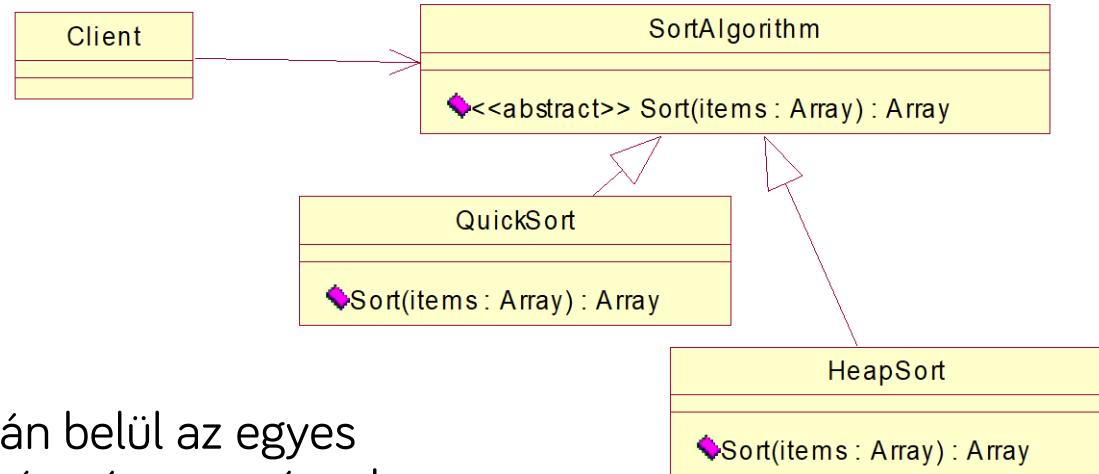


Template method

- Következmények
 - > Lehetővé teszi, hogy az algoritmus invariáns részeit egy helyen definiáljuk, és a változó részeket a leszármazott osztályban adjuk meg.
 - > Így megoldható a kódduplikálás elkerülése: a hierarchiában a közös kódrészleteket a szülő osztályban egy helyen adjuk meg (template method), ami a különböző viselkedést megvalósító egyéb műveleteket hívja meg. Ezeket a “különböző viselkedést megvalósító egyéb műveleteket” a leszármazott osztályban felül kell/lehet definiálni.
 - > Lehetővé teszi ún. hook függvények definiálását: ezek kiterjesztési pontok a kódban.
- Megjegyzés
 - > Kretrendszerek esetében gyakori
 - > A .NET-ben delegate-tel is megoldható a kiterjesztés, C++, Java esetén csak az itt bemutatott (ősben levő virtuális függvény hívása) lehet megoldás
 - > Semmi köze a C++ sablonokhoz (azok generikus típusok)

Strategy

Strategy



- Cél

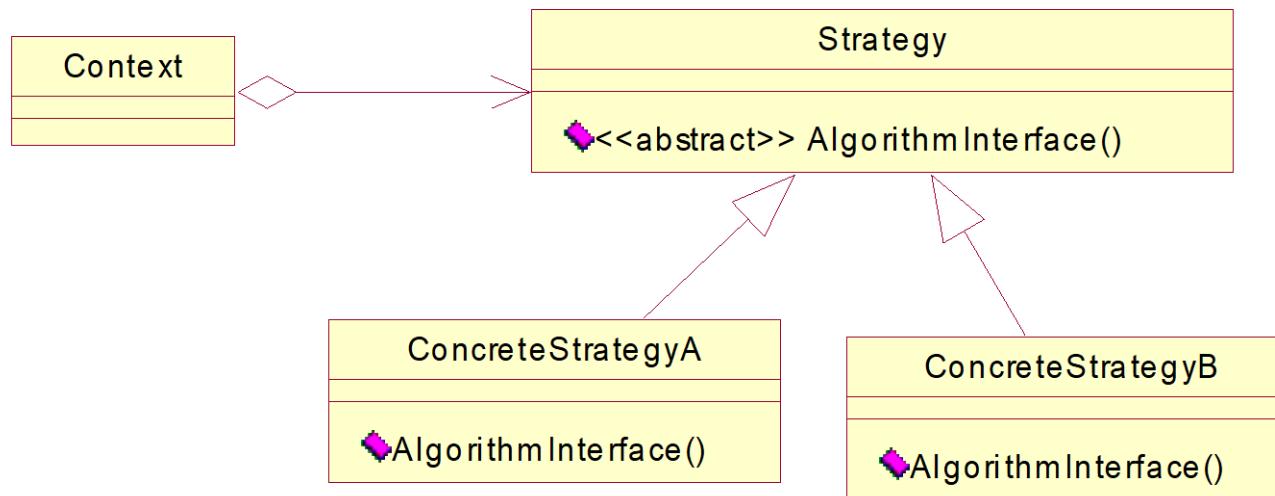
- Algoritmusok egy csoportján belül az egyes algoritmusok egységebe zárása és egymással kicserélhetővé tétele. A kliens szemszögéből az általa használt algoritmusok szabadon kicserélhetők.

- Példa

- A kliens objektum különféle sorrendező algoritmusokat használhat
 - A kliens tartalmaz egy **SortAlgorithm** típusú pointert/referenciát egy konkrét (**QuickSort** v. **HeapSort**) objektumra.
 - A pointer által hivatkozott implementációs objektumot könnyű kicserélni.

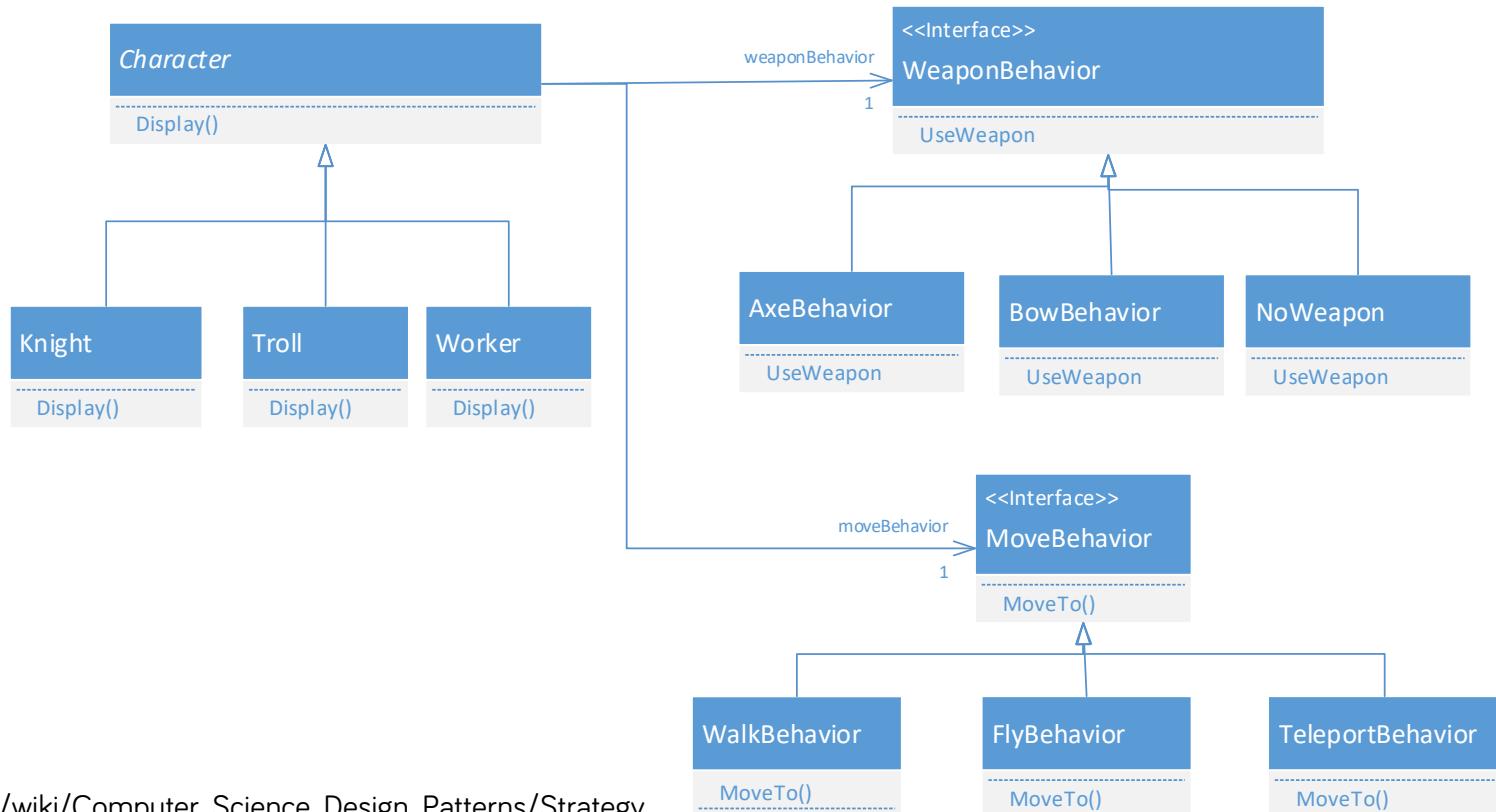
Strategy - Általános osztálydiagram

- Strategy interfésznev mellett szokás a Behavior illetve Policy nevek használata is.



Strategy – szerepjáték „Character” példa

- A minta elve: származtatás/komplex hierarchia helyett az osztály viselkedésének különböző aspektusait kompozícióval tegyük paraméterezhetővé.



https://en.wikibooks.org/wiki/Computer_Science_Design_Patterns/Strategy

Strategy minta, példa magyarázata

- Character:
 - > Egy karaktert reprezentál egy szerepjáték alkalmazásban
- Karakter viselkedésének aspektusai
 - > Fegyverhasználat
 - > Mozgás
 - > Később újabbak is megjelennek
- A viselkedésének minden aspektusa legyen futás közben könnyen megváltoztatható

Strategy minta, példa magyarázata

- Megoldás
 - > minden aspektushoz vezetünk egy külön interfészt (WeaponBehavior és MoveBehavior), és ezekhez megfelelő implementációkat
 - > A Character tartalmaz referenciát minden aspektus implementációra, egy-egy interfészen keresztül (weaponBehavior és moveBehavior asszociáció)
 - > Előnyök
 - Ezen implementációk dinamikusan, futás közben is lecserélhetők.
 - Könnyű új viselkedést vezetni, csak implementálni kell egy megfelelő interfészt, meglévő kódot nem/alig kell módosítani.

Strategized locking

Strategized locking

- A Strategy minta alkalmazása zárolásra
- Környezet
 - Egy osztályt egyszálú és többszálú környezetben is kívánunk használni
 - Többszálú környezetben szükség van zárak alkalmazására

```
public class Cache
{
    private Dictionary<string, object> items =
        new Dictionary<string, object>();

    public object Lookup(string key)
    {
        lock (this)
        {
            // Védendő kód
            return items[key];
        }
    }
}
```

Strategized locking

- De egyszálú környezetben a zárak alkalmazása feleslegesen lassít !!!
- Ekkor egy nem szálbiztos megoldás ez esetben jobb teljesítményt nyújt
- Mi legyen a megoldás?
 - > Írjuk meg mind a szálbiztos és nem szálbiztos verziót? De ki fogja karbantartani? ☹. Nem.
 - > Alternatíva: megírjuk a zárak nélküli osztályt, valamint egy csomagolót, ami az előzőhez szálbiztos hozzáférést biztosít: az használjuk majd, amelyikre éppen szükségünk van.
 - Hátránya lehet: a csomagoló a metódushívások elejétől a végéig zárol (mert nem tud mást), ami lehet felesleges, csak a művelet kis részét kellene zárolni.
 - > Egy jobb alternatíva: Strategized locking minta alkalmazása

Strategized locking

- A minta leírása
 - > Lásd ez előadáshoz tartozó zip mellékletben a „Strategized Locking\StrategizedLocking.cs” példa. A legfontosabb gondolatok:
 - > Bevezetünk egy ILock interfészt, Acquire és Release műveletekkel
 - > Írunk egy NoLock implementációt, melyben az Acquire és Release nem csinál semmit
 - > Írunk egy ObjectLock implementációt, melyben az Acquire objektum szinten zárol, a Release oldja a zárat
 - > A védendő osztályunkba bevezetünk egy ILock típusú (pl. syncObject nevű) tagot, amit egyszálú környezetben egy NoLock, többszálú környezetben egy ObjectLock objektumra állítunk
 - > Ahol a védendő osztályunkban kölcsönös kizáráásra van szükség, használjuk a syncObject tag Acquire és Release műveleteit.

Strategized locking

- Záró gondolatok
 - > A Strategized locking a Strategy minta alkalmazása: egy osztály zárolásra vonatkozó viselkedését tesszük vele paraméterezhetővé.
 - > A Strategy minta alkalmazásával az osztályunk viselkedésének bármely aspektusát egymástól függetlenül paraméterezhetővé tudjuk tenni (zárolás, perzisztencia, stb.)
 - Bármilyen kombinációban használható lesz az osztályunk!!!!

További tervezési minták

- A klasszikus „GoF” minták közül is kimaradt pár:
 - > Prototype, Builder, Bridge, Mediator, Chain of Responsibility, Visitor, Decorator, Iterator, stb.
- Elosztott, konkurens rendszerekre jellemző minták
 - > Szolgáltatás hozzáférés
 - > Konfiguráció
 - > Esemény kezelés
 - > Szinkronizáció
 - > Konkurenencia
- Valós idejű rendszerekre jellemző minták

Összefoglalás

- Tapasztalati tudást hordoznak
 - > Mi is rá tudunk jönni, de
 - Jó sokáig tart
 - Nem jövünk rá
 - Miért ne tanuljunk mások tapasztalataiból
 - > Értékes tudás!
- Célunk
 - > Ismerjünk meg minél több patternet
 - > Hosszútávon legalább arra emlékezzünk, hogy egy adott problémakörben mely patterneket lehet jól használni

Irodalom

- Az interneten bármely minta nevére rákeresve számos leírást találunk, a legtöbb minta a wiki-n is megtalálható.
- [1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: DESIGN PATTERNS, Elements of Reusable Object-oriented Software
- [2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal: A SYSTEM OF PATTERNS, Pattern-Oriented Software Architecture
- Java design patterns:
 - > <http://www.patterndepot.com/put/8/JavaPatterns.htm>
 - > Pdf-ben letölthető!
- <http://www.javacamp.org/designPattern/>

Bináris komponensek, reflexió

Szoftvertechnikák előadás

Benedek Zoltán

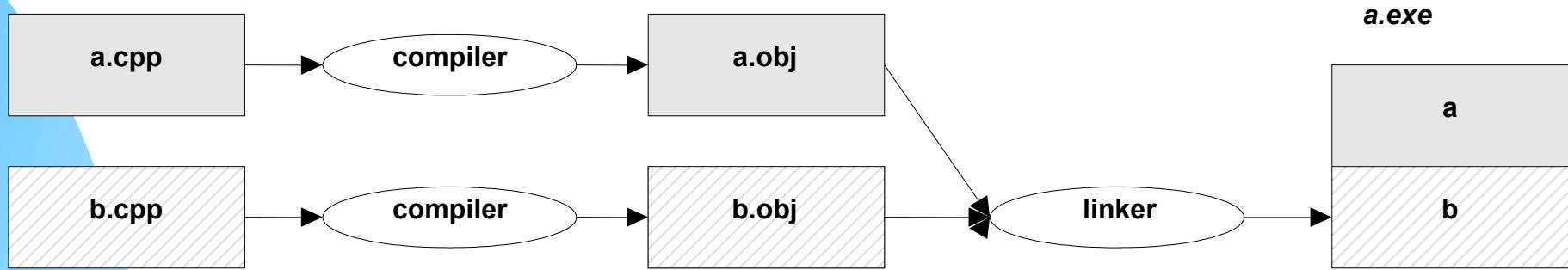
Automatizálási és Alkalmazott Informatikai Tanszék, BME

Tematika

- **Statikus és dinamikus linkelés**
 - ◆ Fogalmak
 - ◆ A fordítás és csatolás (linkelés) folyamata
 - ◆ Statikus linkelés
 - ◆ Dinamikus linkelés
- **Bináris komponensek C++ környezetben**
- **Reflexió .NET környezetben**
 - ◆ Architektúra
 - ◆ Példák

STATIKUS ÉS DINAMIKUS LINKELÉS C, C++ KÖRNYEZETBEN

Fordítás folyamata (ismétlés)



Jellemzők

- ◆ A fordító a forrásfájlokat egyesével dolgozza fel. A kimenet `.obj`, vagy `.o` fájl
- ◆ A linker a címfeloldást végzi

Fogalmak – könyvtár, bináris komponens

■ Könyvtár (**library**) – C, C++, stb.

- ◆ Egy függvénykönyvtár vagy osztálykönyvtár függvénydefiníciókat, osztálydefiníciókat (pl. C++ osztályok), konstansokat és esetleg erőforrásokat tartalmaz **lefordított, bináris** (tehát nem forrás) formában. Ezeket **bináris komponenseknek** nevezzük.

■ Könyvtár felhasználása

- ◆ **include**-olni kell a deklarációkat

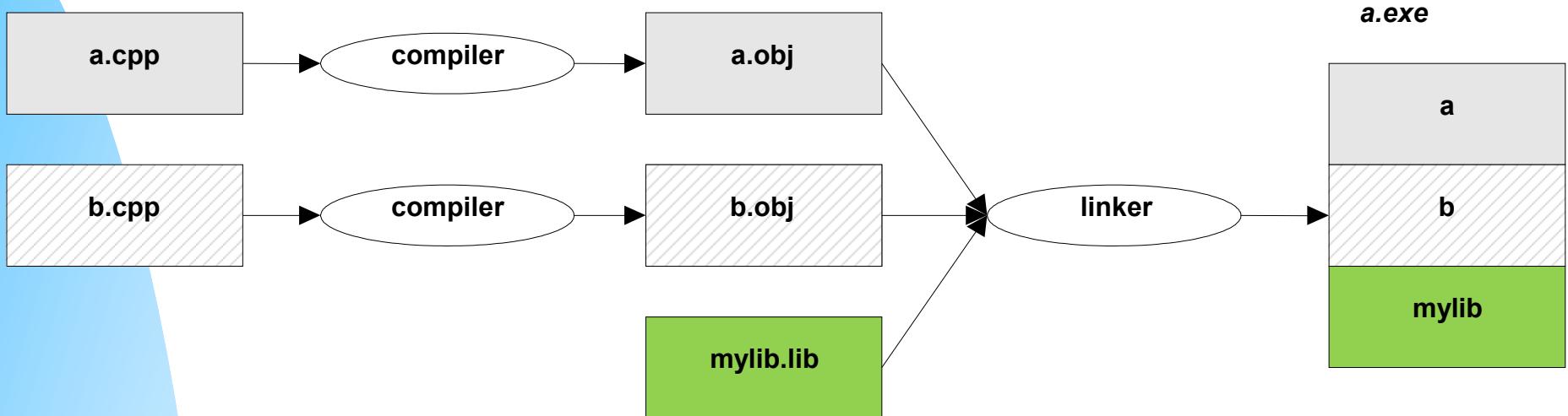
- Pl. printf esetén az stdio.h-t

- ◆ és linkelni kell a definíciókat tartalmazó könyvtárat. Két módon lehet:

- A) **Statikusan**
 - B) **Dinamikusan**

Statikus linkelés

■ Statikus könyvtár felhasználása



■ Jellemzők

- ◆ A könyvtár neve tipikusan `.lib` kiterjesztésű. Ez tulajdonképpen több `.obj` fájl összefűzve.
- ◆ Az alap könyvtári függvények (pl. `printf`) tipikusan a `libc.lib`-ben találhatók. Persze készíthetünk saját statikus library-t is.

Statikus linkelés jellemzők

■ Hátrányok

- ◆ Nagy fájlméret az alkalmazásnál (ahány alkalmazáshoz (.exe) linkelem, annyiszor foglal helyet a háttértáron)
- ◆ Nagy memóriaigény (hiszen ha több futó program is használja ugyanazt a könyvtárat, akkor a statikus linkelés miatt több könyvtárpéldány is lesz a memoriában)
- ◆ Nehézkes hibajavítás: ha esetleg a könyvtárban volt egy hibás függvény, akkor a program javított verzióját csak úgy lehet kiadni, hogy újralinkeljük a programot a javított könyvtárral.

■ Előnyök

- ◆ Önállóan futni képes az alkalmazás, nincs szükség külső könyvtárfájlokra (ennél fogva nincsenek verzió problémák, egyszerű az installálás, stb.).

■ Demo

- ◆ „0 Static Library\Windows” példa

Dinamikus linkelés

- **Definíció: a futó program csak futásidőben tölti be a számára szükséges könyvtárakat**
 - ◆ A program könyvtárfüggvény hivatkozásait futásidőben kell feloldani.
 - ◆ Windows alatt .dll (Dynamic Link Library), Linux alatt .so kiterjesztés.

Dinamikus linkelés jellemzők

■ Előnyök

- ◆ Ha több process használja a könyvtárat, akkor a memóriába csak egy példányban töltődik be (a kódrész, az adat marad külön!)
- ◆ A háttértáron is csak egyszer foglal helyet (ha az alkalmazások megosztottan használják pl. a windows\system könyvtárból)
- ◆ Kis alkalmazás (.exe) fájlméret
- ◆ Ha a .dll-ben szerepel egy hibás függvény, akkor a programot egyszerűen a .dll cseréjével is javíthatjuk, nem kell újrafordítani.

Dinamikus linkelés jellemzők

■ Hátrányok

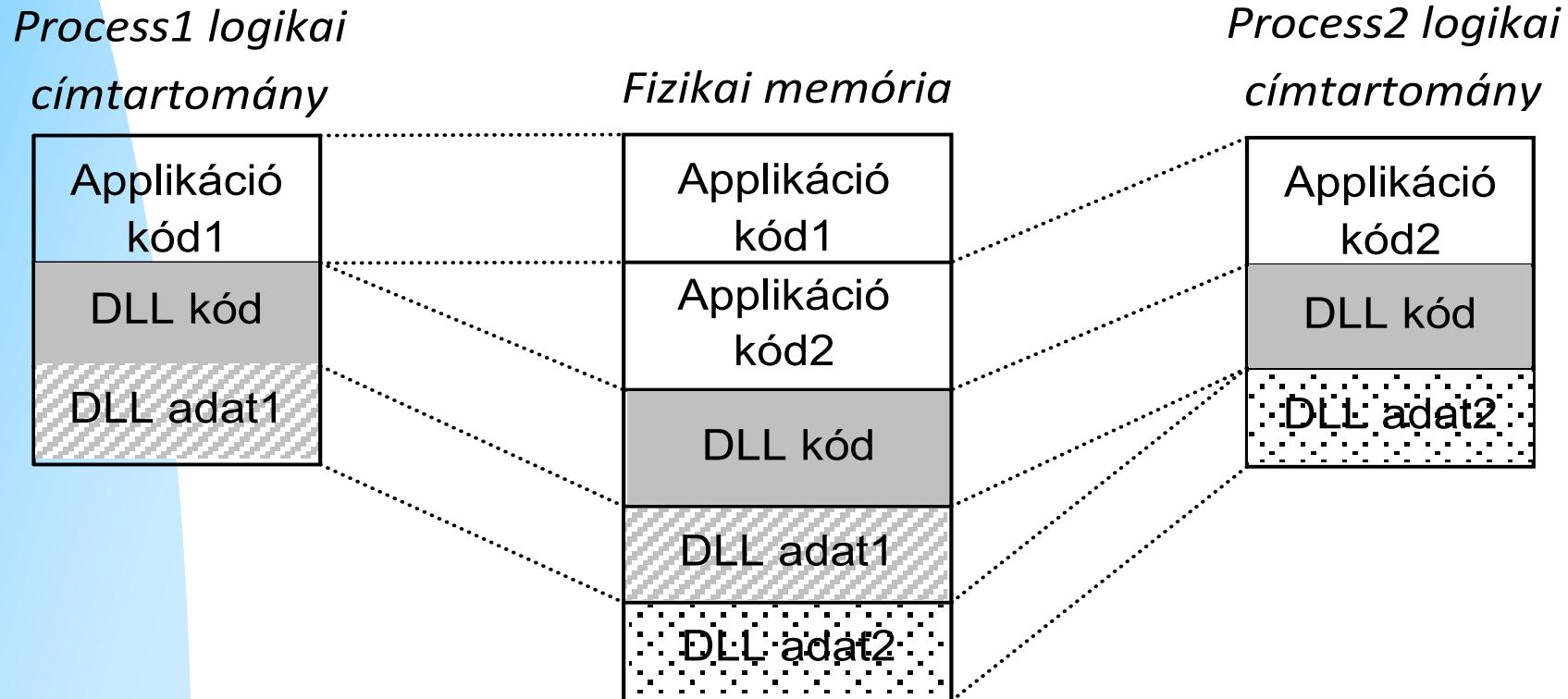
- ◆ A program futásakor jelen kell legyenek az általa használt DLL-ek, ezek nélkül az alkalmazás futásképtelen.
- ◆ Ha az alkalmazások megosztottan használják (pl. a windows\system könyvtárból): verzió kavarodás, DLL hell.
(DLL hell példa volt a .NET szerelvényeknél)

■ A Windows a következő helyeken keresi a DLL-eket

- ◆ az exe-t tartalmazó könyvtár
- ◆ az aktuális könyvtár
- ◆ <windows>\<system32> könyvtár
- ◆ <windows> könyvtár
- ◆ a PATH -ban felsorolt helyen

DLL kód megosztása a memóriában

- DLL kódja csak egyszer töltődik be a memóriába, DLL adat terület viszont annyiszor, ahány processz használja az adott pillanatban a DLL-t. Csak így lehetnek védettek egymással szemben a folyamatok!





BINÁRIS KOMPONENSEK C, C++ KÖRNYEZETBEN

Bináris C++ komponensek (DLL) verziókezelése

- T.f.h. van egy komponensünk (pl. exe, de lehet dll is), mely használ egy dll-t: pl. meghívja a dll-ben levő valamely osztály valamely metódusát.
- Ha a dll-ben levő osztályok szerkezete módosul (pl. felveszünk egy privát tagváltozót), **akkor a használó komponenst is újra kell fordítani!** Akkor is, ha a felhasznált osztályok interfésze (publikus tagok) nem változnak!
- Ez nagyon megnehezíti a többkomponensű alkalmazások fejlesztését, verziózását!
- **A problémára van C++ nyelvben is megoldás („interfészek”, vagyis csupa absztrakt műveleteket tartalmazó osztályok alkalmazása).**
 - ◆ De ez nagyon nehézkes. A C++ nyelv korlátozottan alkalmas komponensek fejlesztésére.
- A modern környezetekben (.NET, Java) ez sokkal egyszerűbb, a fenti probléma soha nem áll fent (nem kell a felhasználó komponenst újra fordítani), mert a komponensek nem függenek a más komponensekben levő osztályok belső szerkezetétől.



BINÁRIS KOMPONENSEK .NET KÖRNYEZETBEN

Bináris komponensek .NET környezetben

- .NET környezetben egy DLL-t vagy egy EXE fájlt tekintünk komponensnek (vagyis egy szerelvényt/assembly-t)
- Visual Studio alatt az egyes projektek kimente általában egy-egy komponens
- Ha egy DLL-ben levő osztály szerkezete módosul (de az interfésze nem), akkor a DLL-t használó komponenseket - a C++ nyelvvel ellentében - NEM kell újra fordítani. Így sokkal egyszerűbb a többkomponensű alkalmazások fejlesztése.
- Ez azért lehetséges, mert a komponensek nem függenek a más komponensekben levő osztályok belső szerkezetétől.

REFLEXIÓS TECHNIKÁK

Reflexió

■ Reflexió (reflection)

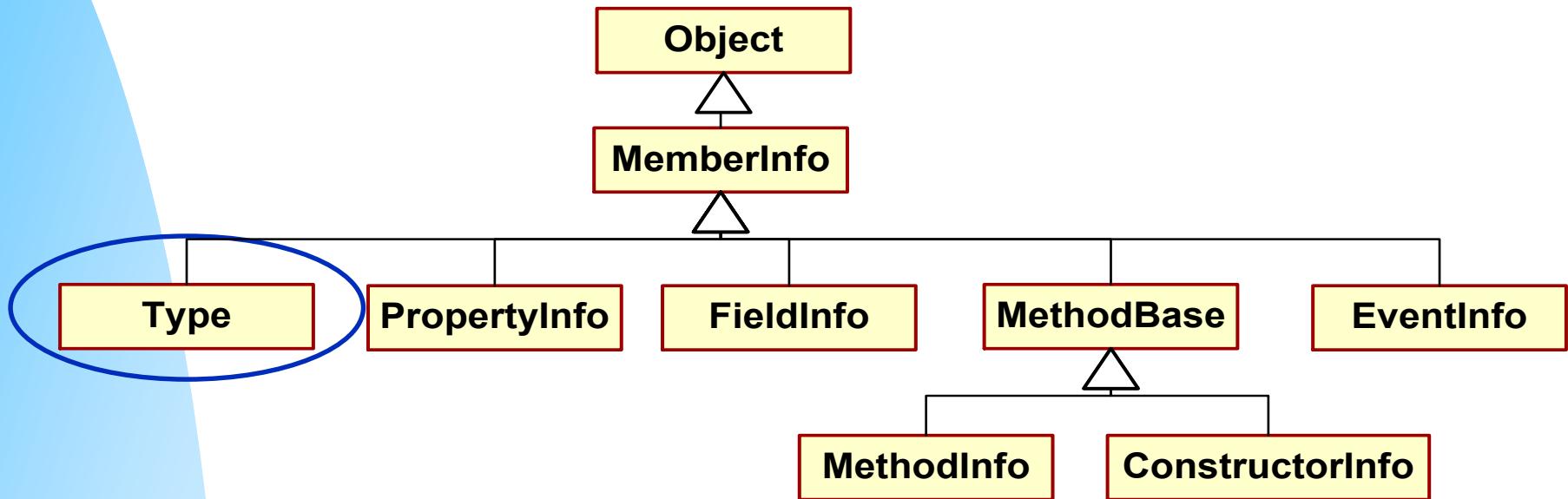
- ◆ Elsősorban szerelvények és típusok meta-adatait kérdezhetjük le.

■ Részletesebben

- ◆ Lekérdezhetjük, hogy egy szerelvényben milyen típusok vannak.
- ◆ Lekérdezhetjük: az egyes típusok (osztályok, interfések, stb.) felépítését: pl. tagváltozók, tagfüggvények, event-ek, stb. listája.
 - Be is állíthatjuk a tagváltoozók értékét
 - Meg is hívhatjuk az egyes metódusokat
- ◆ Példányosíthatunk objektumokat úgy, hogy azok típusa fordításkor még nem ismert (futás közben egy sztring formájában tudjuk a típust megadni)
- ◆ Lekérdezhetjük az egyes nyelvi elemekhez (osztályok, tagjaik, stb.) tartozó attribútumokat. Lehetőség van akár saját attribútumok használatára is.

Reflexió

- **Type** osztály – egy típust reprezentál

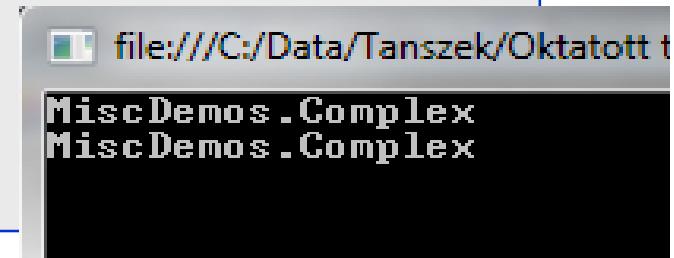


- Demók
 - ◆ „3 Reflection” mappa „MiscDemos” projekt

Reflexió

■ Példa egy objektum, illetve osztály típusának lekérdezésére

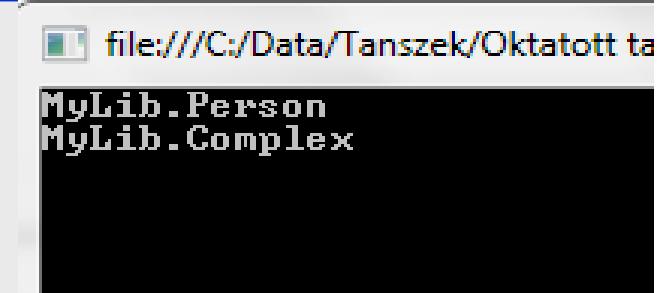
```
Complex c1 = new Complex(10, 10);
Type t1 = c1.GetType(); // A GetType az Object-ben definiált
Console.WriteLine(t1.FullName);
// A typeof egy C# operátor
Type t2 = typeof(Complex);
Console.WriteLine(t2.FullName);
```



- ◆ A Type.FullName a típus nevét adja vissza névtérrel együtt

■ Példa – Írjuk ki a MyLib.dll szerelvényben levő típusok listáját

```
Assembly assembly;
assembly = Assembly.LoadFrom("MyLib.dll");
Type[] types = assembly.GetTypes();
foreach (Type type in types)
    Console.WriteLine(type.FullName);
```

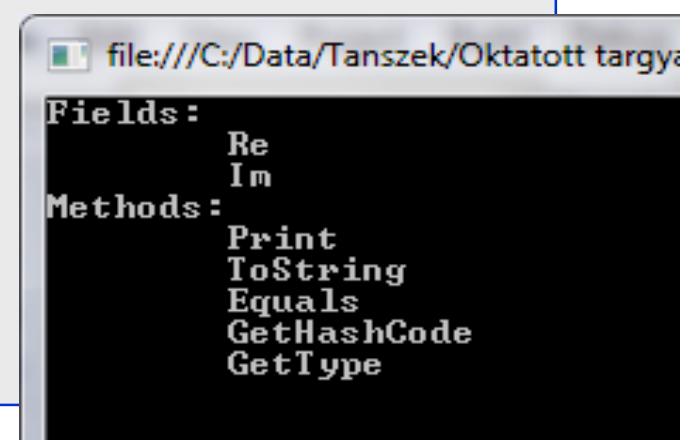


- ◆ Az Assembly típus egy betöltött szerelvényt reprezentál (.dll, .exe)

Reflexió

■ Példa osztály tagváltozóinak és tagfüggvényeinek lekérdezése

```
Type type = typeof (Complex);  
Console.WriteLine("Fields:");  
foreach (FieldInfo fi in type.GetFields())  
    Console.WriteLine("\t" + fi.Name);  
Console.WriteLine("Methods:");  
foreach (MethodInfo mi in type.GetMethods())  
    Console.WriteLine("\t" + mi.Name);
```



- ◆ A Print metódus a Complex osztályban definiált, a többi az Object ősből örökölt.

■ FieldInfo tagok pl.:

- ◆ Name – a tagváltozó neve
- ◆ DeclaringType
- ◆ IsPublic
- ◆ MemberType – a tagváltozó típusa

...

Reflexió (nem kell tudni a példát)

■ Példa: objektum példányosítása, tagok elérése

```
// Egy Complex objektum létrehozása
Type complexType = Type.GetType("MiscDemos.Complex");
ConstructorInfo ci = complexType.GetConstructor(new Type[0]);
Object instance = ci.Invoke(null);

// A „Re” mező lekérdezése
FieldInfo fi = complexType.GetField("Re");
Console.WriteLine(fi.GetValue(instance));
// A „Re” mező beállítása
fi.SetValue(instance, 10);
// A Print hívása
MethodInfo mi = complexType.GetMethod("Print");
mi.Invoke(instance, null);
```

■ Jellemzők

- ◆ Rendkívül rugalmas, mert a tagok nevét sztringben adhatjuk meg.
- ◆ Nagyon lassú!

Reflexió – attribútum példa

■ Reflexiós példa attribútumra

- ◆ Példa erre: saját attribútum létrehozása, használata és lekérdezése

A feladat:

- ◆ Írunk egy olyan osztályt, amely képes bármilyen osztály objektumainak adatfolyamba (pl. fájlba) mentésére.
- ◆ Biztosítsunk lehetőséget arra, hogy csak az általunk meghatározott osztályok objektumai, illetve azok meghatározott tagváltozói kerüljenek mentésre. Az, hogy egy tagváltozó milyen néven kerüljön mentésre, szintén szabályozható legyen.

A megoldás alapelve:

- ◆ Definiálunk egy **StorableClass** nevű attribútumot: csak azon osztályokat mentjük, melyek el vannak látva ezzel az attribútummal.
- ◆ Definiálunk egy **Storable** nevű attribútumot: csak azon tagváltozókat mentjük, melyek el vannak látva ezzel az attribútummal. Az attribútum egy paramétere, hogy milyen néven kell az attributált tagváltozót elmenteni.

Reflexió – attribútum példa

Demo

- „3 Reflection” mappa „GeneralSave” projekt

Így mentünk

```
Student student = new Student("James Bond", "007007");  
SaveUtils.Save(student); // Elmenti a student objektumot
```

Így szeretnénk szabályozni attribútumokkal a mentést

```
[StorableClass]  
class Student  
{  
    [Storable("Name")]  
    private string name;  
    [Storable("Neptun", SaveType= true)]  
    private string neptun;  
    ...  
}
```

Reflexió – attribútum példa

■ StorableClass és Storable attribútumok definiálása

```
[AttributeUsage(AttributeTargets.Class)]  
class StorableClassAttribute: System.Attribute {  
    public StorableClassAttribute() { }  
}
```

```
[AttributeUsage(AttributeTargets.Field)]  
public class StorableAttribute: System.Attribute  
{  
    string name;  
  
    public StorableAttribute(string name)  
    {  
        this.name = name;  
    }  
  
    public string Name { get { return name; } }  
}
```

- ◆ A System.Attribute osztályból kell egy saját osztályt leszármaztatni
- ◆ Az AttributeUsage azt határozza meg, mihez tudjuk az adott attribútumot hozzárendelni

Reflexió – attribútum példa: mentés

A mentés implementációja

```
class SaveUtils
{
    public static void Save(object o)
    {
        Type type = o.GetType();

        // Ha nincs StorableClassAttribute az osztályhoz rendelve, akkor nem mentünk
        object[] attributes =
            type.GetCustomAttributes(typeof(StorableAttribute), false);
        if (attributes.Length == 0)
            return;

        FieldInfo[] fieldInfos = type.GetFields(BindingFlags.Public | BindingFlags.NonPublic
                                                | BindingFlags.Instance);
        foreach (FieldInfo fi in fieldInfos)
        {
            object[] fieldAttributes = fi.GetCustomAttributes(
                Type.GetType("Storage.StorableAttribute"), false);
            if (fieldAttributes.Length == 0)
                continue;
            StorableAttribute attr = (StorableAttribute)fieldAttributes[0];

            // Ebben a demóban nem mentünk, helyette kiírjuk a konzolra a mentés paramétereit.
            Console.WriteLine("Name: {0}", attr.Name);
            Console.WriteLine("Value: {0}", fi.GetValue(o).ToString());
        }
    }
}
```

Ellenőrző kérdések

- Adja meg a bináris komponens definícióját!
- Ismertesse a statikus linkelést, adja meg az előnyeit és hátrányait (a dinamikus linkeléssel szemben)!
- Ismertesse a dinamikus linkelést! Adja meg az előnyeit és hátrányait (a statikus linkeléssel szemben)!
- A C++ nyelv miért alkalmas korlátozottan komponensek fejlesztésére!
- Egy példa segítségével mutassa be, hogyan lehet C++ nyelvben megoldani a verziózásból adódó alapvető problémát!
- Definiálja a reflexió (reflection) fogalmát! Milyen szolgáltatásokat nyújt a mai modern fejlesztőkörnyezetekben?
- Mutasson példát egy objektum, illetve osztály típusának lekérdezésére!
- Mutasson példát egy szerelvényben levő típusok listázására!
- Mutasson példát egy osztály tagváltozónak és tagfüggvényének listázására!
- Mutasson példát saját attribútum létrehozására, használatára és lekérdezésére!