Szoftvertechnikák ZH összefoglaló

Mi lesz ZH-n?

Fejezetek

- A modell és a kód kapcsolata, interfész és absztrakt ősosztály
- Futtatókörnyezetek
- Modern nyelvi eszközök
- Win32 alkalmazások architektúrája és eseményvezérelt programozás
- Felügyelt vastagkliens alkalmazások fejlesztése .Net-tel
- Saját vezérlő készítése
- Grafika
- Generikus Típusok
- Multithreading
- Bináris komponenesek, reflexió
- Adatbázis
- Architekturális tervezés
- Letöltés

Minta ZH

- ZH_Szoftech2007tavasz_120204120055.doc

ZH típusa

- Gyakorlat- és példacentrikusabb, többet kell papíron programozni
- Nem algoritmusokat pontoznak, hanem a nyelvi elemeket, magyarázattal
- Javítás során szintaktikai hibák nem lényegesek (pontosvessző, függvényparaméterek sorrendje), szemantikai annál inkább (ha látszik, hogy nem érted a működés elvét)!
- A ZH három feladata **beugró**
    - Eseményvezérelt programozás témakör
        - Console és Windows Forms → **gyakorlati** példa
        - WIN32 (natív) és C# → **elméleti** kérdések
    - Modern nyelvi eszközök → **gyakorlat** és **elmélet**
        - Property
        - Delegate
        - Event
        - Attribute
        - Referencia-, Érték-típusok
    - *Szálkezelés* (nem biztos, csak nagy valószínűséggel, de mindenképpen olyan ami volt gyakorlaton is!)

ZH témakörei

- Első témakörökhöz tartozó feladatok, Pl.:
    - Eddig property/delegate/event témakört példakóddal kellett bemutatni
    - Most konkrét feladatot kell megvalósítani Pl:
        - sorrendezés delegate segítségével
        - hírszerver megvalósítása események segítségével
        - stb.

- Windows Forms feladat, Pl.:
  - Billentyű-, egér-esemény
  - Időzítő alkalmazása
  - Rajzolás
  - stb.
- Szálkezelés, Pl.:
  - Szálak
  - Zárak
  - ManualResetEvent / AutoResetEvent
  - stb.
- Elméleti kérdések bármely témakörből

Mit kell hozni ZH-ra?

- 8 db üres lap
- íróeszköz
- igazolvány
- uzsonna
- jókedv

Terembeosztás

- Szoftech/2015/eredmények

A modell és a kód kapcsolata, interfész és absztrakt ősosztály

Objektum

- egységbe zárja az adatot és rajta dolgozó műveleteket
- minden objektum entitás, megkülönböztethető a többi objektumtól
- állapota van
- (saját) adatot rejt

Osztály

- azonos objektumok egy osztályba tartoznak
- minden objektum tudja mi az osztálya

Osztályok közötti kapcsolatok

- asszociáció, általánosítás, specializáció, aggregáció, függőség (szofttech anyag)

Implementáció

- forward engineering: modellből kód
- reverse engineering: kódból modell
- round-trip engineering: előző kettő vegyesen
  - a modell és a kód végig szinkronban van

Interfész

- felület amin keresztül egy szolgáltatást elérünk
- pontosan megfogalmazva: egymással szemantikailag szoros kapcsolatban lévő műveletek halmaza
- előnye: kliens pl. csak egy kiszolgáló interfészt ismer de több kiszolgálót tud vele használni (mint Javában a Collection és megvalósításai)

Futtatókörnyezetek

Példa

- Java (JVM)
- .NET (CLR)

Virtuális gépekről általában

- absztrakt számítógép architektúra
- szoftverként működik a hardver fölött
- multiplatform
- felügyeli a kódot és az adatot
    - szemétgyűjtés, memória kezelése
    - biztonságos
- IL (intermediate language)
    - ha létrehozunk egy átmeneti nyelvet, O(nm) helyett O(n+m) fordítóra van szükség
    - pl. C#, F#, J#, L#, P# → ARM, x86, PowerPC = 15 fordító,
      de C#, F#, J#, L#, P# → IL → ARM, x86, PowerPC = 5 IL-re fordító + 3 interpreter/JIT
- egyes esetekben kompaktabb mint a natív kód
- tud skálázódni az aktuális platformon
- más nyelvek támogatása (natív dll meghívása)

.NET

- Framework azaz a .NET keretrendszere
    - common language runtime – CLR (ez futtatja a .NET-es nyelveket)
    - osztálykönyvtárak – BCL (base class library)
- Technológiák a nyelvben
    - ASP.NET – webes alkalmazásokhoz
    - Windows Forms – vastag kliens alkalmazásokhoz
    - ADO.NET – adatkezelés (elavult)
    - Web services, stb.
- nyelvfüggetlen, minden rá épülő nyelv hozzáférhet .NET objektumokhoz

CLR

- közös nyelvi futtatókörnyezet
- IL, GC, CAS
- objektumorientált
- keretrendszer beépített osztályai névterekbe rendezve
- egységes, sokféle típust definiál
- integráció régebbi technológiákkal
    - COM, C++, ActiveX
    - Win32 API elérhető (Pinvoke())
- része a felügyelt C++ azaz C++ CLR, ahol vegyül a natív és a .NET-es kód

C#

- nincs mutató csak referencia
- ez a nyelv mutatja meg a legjobban a .NET képességeit
- minden objektumként működik, nincs POD, mint C++-ban pl. 5.ToString()
- garbage collector – nincs delete/memory leak

    - tulajdonság: a fiatalabb objektumok hamarabb felszabadulnak
    - hátrány: nem determinisztikus
    - vészhelyzetben: System.GC.Collect()
- felügyelt a kód
    - metainformációk az objektumokhoz: `new Object().GetType().GetMethods()`
    - nem lehet hülyeséget castolni, kivételt dob a program
    - tkp. nem lehet túl súlyos szemantikai hibát véteni hibaüzenet nélkül
- egységes hibakezelés

Felügyelt kód

- forrás kód → CLR fordító → IL + metaadatok → JIT fordító → natív kód
- IL
    - processzor és architektúra független
    - ellenőrizhető referenciák, típusok, hívási verem
    - továbbfordításra (JIT, bin) tervezték nem interpretálása
    - objektumorientált
    - könnyű visszafejteni
- assembly
    - általában egy dll vagy exe
    - IL kódot és erőforrásokat (jpg, txt, stb.) tartalmaz
    - metaadatokat tartalmaz az IL kódról, .NET osztályokról
    - itt egy assembly fájl van, nem úgy mint a javában (*.class)
    - metaadatok
        - név (fájlnév)
        - verzió
        - assembly referenciák más assembly-kre
        - importált modulok listája, stb.
        - azonosított assembly-k
            - nincs dll hell mert pontos verzióra hivatkozik a program
            - digitális aláírás
- CLR biztonság
    - felhasználói jog alapú biztonság
    - program eredete alapján védelem
    - beállíthatjuk a hálózatok milyen (windows) user/group férhet hozzá a programhoz
    - program jogosultságainak kezelése

Modern nyelvi eszközök

Nyelvi elemek

- megkülönböztetés
    - érték: int, float, enum, struct
    - referencia: interface, class, array, string
- alap típusok kb ugyanazok mint C-ben, bool nem cserélhető fel int-tel
- minden típus a System.Object-ből származik
    - metódusok: ToString, GetHashCode, Equals, GetType, stb.
- érték típusok stack-en jönnek létre (mivel int a = 5)
    - sokkal gyorsabb mint a referencia típus
    - nem lehet örökölni velük

    - interfészt viszont implementálhat (struct)
    - pl. DateTime, Decimal, enum, Complex
- referencia típusok heap-en (mivel Object a = new Object())
    - nagy teljesítménycsökkenés, ha GC takarítja őket
    - ne használjunk soha destruktort (finalizert)
    - minél előbb engedjük el: using statement vagy obj = null
- System.String
    - nem megváltoztatható az értéke
    - minden tagfüggvénye új példányt ad vissza (a GC a régit eltakarítja)
    - ha sokat kell konkatenálni, System.Text.StringBuilder/IO.StringWriter
- Új statement-ek:
    - foreach(<type> <varname> in <iterable >){ ... }
    - lock(<object>){ ... }; (Javában synchronized block)
    - checked {}, unchecked {} – airtmetikai műveletek hibakezelése
- Osztályok
    - több interface-t implementálhat
    - csak egy osztálytól örökölhet
- **Property – kell tudni!**
    - van egy tagváltozónk
    - írhatunk hozzá syntactic sugar get/set-et:

```
private int name;
public int Name {
    get { return name; }
    set { name = value; }
}
```

    - nem lehet ugyanaz a neve a property-nek és a tagváltozónak! (nagybetű konvenció)
- Indexer
    - tároló építéséhez
    - megvalósításra olyan mint a Property
    - elméletben olyan mint az operátor túlterhelés
    - a [ ] operátorral get/set-elheted az osztályt mint tárolót (ha van tárolója)
- Reflection
    - is: típus tesztelés (pl. int is int == true)
    - as: castolás, ha nem lehetséges, null lesz
    - typeof: visszaadja a változó típusát (System.Type-ként)
- Operátor túlterhelés
    - Speciális operátorok
        - sizeof, new, is, (typeof)
    - tulajdonképpen úgy működik mint C++-ban
- **Delegate – kell tudni!**
    - nincs függvénypointer C#-ban
    - működése:
        - definiálunk egy delegate-et:
          public delegate void FunctionWrapper()
        - ahol várjuk a függvényt, beírjuk a delegate nevét
          (függvény paraméter, event, stb.)
        - használjuk:
          feldolgozó_függvény(delegate_neve(saját_függvény))

        - Példakód:

```
// ez a wrapper
delegate string FunctionWrapper();
// ez a függvény kapja meg a callback-et
void receiver(FunctionWrapper callback){
    Console.Write(callback());
}
// ez a callback
string myFunction(){
    return "Hello";
}
public void test(){
    // itt használjuk, kiírja, hogy "Hello"
    receiver(myFunction);
}
```

- **Event – kell tudni!**
    - delegate-ekre épített nyelvi elem
    - event-et csak az event-et közvetítő osztályból lehet hívni
    - beregisztrálás az event-hez való hozzáadással lehet
    - Példakód:

```
// ez a wrapper
delegate string FunctionWrapper();
// ez az event
event FunctionWrapper OnEvent;
// ez az event handler
string myEventHandler() { return "Hello"; }
// így kell használni
void test(){
       OnEvent += myEventHandler;
}
```

- Struct
    - adat és kód
    - olyan mint az osztály, de nincs öröklés
    - mindig érték szerint van átadva
- Tömbök
    - érték típusnál lefoglalja őket azonnal
    - referencia típusnál végig kell rajtuk iterálni és meghívni a konstruktort
- Virtual method
    - explicit ki kell írni az őt override-oló metódus elé, hogy override
    - new virtual: újradefiniálod a függvényt, csak a neve marad
- Hozzáférés
    - public, protected, private maradt
    - sealed: nem lehet származni
    - internal: publikus elérés assembly-n keresztül
    - protected internal: védett assembly-n belül és származtatott típusra(?)
- Cast-olás
    - kivételt dob, ha nem lehet
    - as: null-t ad vissza
    - is
- Attribútumok
    - nem az osztály része, inkább információ a fordítónak
    - futási időben lekérdezhetőek

    - saját attribútumokat is csinálhatunk
    - pl. [Serializable], [NonSerialized]
- Konstans
    - const: mint C++-ban a constexpr
    - readonly: Java final, C++ const
- statikus konstruktorok
    - egy osztályon belül lehetnek statikus attribútumok és metódusok
      ilyenkor a statikus konstruktor inicializálja az attribútumokat
    - akkor fut le, amikor először hozzá akarunk nyúlni az osztály statikus részéhez
- destruktorok
    - nem determinisztikus
    - nem tudjuk az objektum felszabadítás sorrendjét, idejét, stb.
    - implementálja az Idisposable interfészt
- volatile
    - lásd C++
- ref / out
    - függvény paraméter lehet ref vagy out
    - ref: referencia szerint kerül átadásra
    - out: nem kell inicializálni, a függvény ad értéket neki
    - boxing: érték változót dobozolunk object-be, hogy referenciává váljon
- exception handling
    - try {} catch () finally {} – Java
- namespace
    - lásd C++, de nem :: hanem .

Win32 alkalmazások architektúrája és eseményvezérelt programozás

WinAPI

- Változók kicsit más nevűek mint az alap nyelven (unsigned int: UINT)
- hungarian notation-t használnak (b: char, n: int, p: pointer, lp: far pointer)
- rengeteg preprocesszor konstans van (pl. `WM_CLOSE` --> **W**indow**M**essage**_CLOSE**)
- függvényhívások a stdcall konvenciót követik
- ha létrehozunk valamit az API-val, általában handle-t kapunk. (pl. HWND, HANDLE, HFONT)

Eseményvezérelt programok WinAPI-val

- olyan mint egy állapotgép
- a program indulása után belefut egy while ciklusba ami folyamatosan vár a GetMessage függvényre
- a GetMessage visszatér egy üzenettel, ha valamit csináltunk (mozgattuk az egeret, nyomogattuk a billentyűzetet, stb.)
- ezt az üzenetet feldolgozza a WndProc nevű függvény egy nagy switch-ben.
  pl. `WM_MBUTTONDOWN` azaz lenyomtuk az egér egyik gombját

Felügyelt vastagkliens alkalmazások fejlesztése .Net-tel

Kitérő

- az osztályok kaphatnak egy partial kulcsszót, ami megengedi, hogy úgy használjuk a projekten belül, mintha namespace lenne és különböző helyeken deklarálunk benne dolgokat

    - erre azért van szükség, mert winforms-ban a designer rész is hozzáad, meg mi is szeretnénk saját függvényeket írni. de hogy ne keverjük össze a kettőt, két külön fájlba megy az osztály.

Alkalmazás architektúra

- System.Windows.Forms névtér
- Minden ablak egy Form-ból származik

Windows Forms

- sok tulajdonságot, eseményt definiál
- event-ek: Load, Click, Resize, Move, KeyDown
- vezérlőelemeket helyezhetünk el: ezek a komponensek (pl. TextBox, Label, stb.)
    - ezek lesznek a Form tagváltozói
    - InitializeComponent()-ben példányosodnak
    - ők is sok tulajdonságot és eseményt definiálnak
- A Windows Forms ráépül a natív üzenet- és ablakkezelésre
    - HWND megfelelője a Control.Handle
    - csomagolóként működik a natív ablak körül
    - üzenetkezelés is megvan benne: Application.DoEvents()
- menubar, toolbar, statusbar, context menu

Időzítők

- System.Windows.Forms.Timer komponens
    - periodikus, időzített esemény
    - intervallumot adunk meg
    - tiltás/engedélyezés
    - Start(), Stop()
    - Pontatlan (~20ms)

Dialog Windows

- beállításokhoz, információs doboz, stb.
- modális megjelenítés (nem válthatsz vissza a mögötte lévő (parent) ablakra)
- DialogResult-ban az ablakból való visszatérés értéke

Párbeszédablakok

- ??? igen, vannak

Fontosabb alap vezérlők

- Button, CheckBox, ComboBox, DateTimePicker, Label, ListBox, ListView, DataGridView, RadioButton, TextBox, TreeView

Fontosabb tároló vezérlők

- GroupBox, Panel, SplitContainer
- TabControl – TabPage vezérlők adhatók hozzá

Vezérlő hierarchia

- Component
    - nem feltétlenül vizuális, designerben használható komponens Formhoz
- Control

    - Minden vezérlő őse
    - Egy natív ablak tartozik alá (Control.Handle)
    - összes közös tulajdonság, esemény, művelet hozzátartozik
- ContainerControl
    - Tároló vezérlők számára készült
    - Ha más Form-ra kerül a fókusz, megőrzi a tulajdonságait (TextBox, stb.)
- szülő-gyerek viszony
    - Control.Controls: gyerekablakok
    - gyerek ablak a szülő része, olyan mintha fizikailag benne lenne
    - a szülő ablak a Parent tagváltozóból érhető el
- birtokos-birtokolt viszony
    - lazább mint a szülő-gyerek
    - z-order-ben a birtokolt ablak a birtokos előbb helyezkedik el
    - birtokolt: Form.OwnedForms mutatja
    - birtokos: Form.Owner mutatja

Form és események

- delegate típus Windows Forms-nál általában:
  ```
  public delegate void EventHandler(Object sender, EventArgs e)
  ```
    - sender: esemény kiváltója
    - EventArgs: esemény paraméterei
        - EventArgs nem hordoz információt
        - Ebből kell leszármazni, ha van esemény paraméter
- Billentyű események (Control event-ek)
    - KeyDown – KeyEventHandler – {Shift, KeyCode, KeyData}
    - KeyUp – ugyanaz
    - KeyPress – teljes gombnyomás – {KeyChar}

Destruktor

- ha egy dll-t unload-olunk
- ha drága erőforrást használunk pl. natív ablak, adatbázis, file, lock
- Mit használjunk, ha a destruktor nem determinisztrikus?
    - IDisposable interfész
    - void Dispose() metódus
        - ebben szabadítsuk fel az erőforrásokat
    - destruktorban is hívjuk meg a Dispose-t
- IDisposable miatt using statement használata az API osztályain (ők tuti támogatják)
- Tehát amíg a destruktorral explicit megadhatjuk, mit töröljön a GC, ha éppen gyűjt, addig a Dispose csak egységes kezelést biztosít a nem felügyelt erőforrások determinisztikus felszabadításához

Összefoglalás

- egyszerű, de a WPF jobb

Saját vezérlő készítése

Három módszer

Control osztályból származás

- akkor használjuk, ha teljesen új vezérlőelemet készítünk

- emiatt csak az általános Control-ra vonatkozó tulajdonságokat kapjuk meg
- természetesen hozzáadhatunk saját property-t, eseményt
- rajzolás is a mi feladatunk
- pl. aktuális időt mutató label

Vezérlőből származás

- létező vezérlőt akarunk testreszabni (speciális viselkedés megvalósítása)
- pl. TextBox-ból PasswordBox

UserControl készítése

- mivel a UserControl tartalmazhat más UserControl-okat, lehet a miénken belül is pl. TextBox
- példa: FilePicker – TextBox és mellette egy Button, melyre a OpenFileDialog jön elő
- összetett, moduláris felület tervezéséhez

Grafika

GDI

- graphical devices interface – grafikus megjelenítést tesz lehetővé Win32 alkalmazásokhoz
- képességek:
    - vektorgrafika
    - 2D
    - szöveg, bitmap megjelenítés
    - kép transzformációk
- eszközfüggetlen grafikus megjelenítés – alapja a DC (Device Context)
    - grafikus eszközt reprezentál
    - HDC (HandleDC)-t kapunk, menete:
        - eszközkapcsolat létrehozása
        - rajzolása az eszközre
        - eszközkapcsolat lezárása
    - bármire rajzolhatunk; fejléc, menü, képernyő, memória, nyomtató
    - ha rajzoltunk és valami felé kerül, nem jegyzi meg, frissítenünk kéne
    - WM_PAINT flag - újrarajzolás

GDI+

- olyan mint a GDI, de .NET
- PaintEventArgs – paraméter event-nél
- PaintEventArgs.Graphics – ez a HDC
- PaintEventArgs.ClipRectangle – terület a Graphics-en
- Form.ClientRectangle – ablak területe
- <component>.ClientRectangle – akármire rajzolhatunk
- színkezelés
    - Color.Red – piros szín (wow)
    - Color.FromArgb() – alpha + rgb
    - Color.Empty – nincs kitöltés, null
    - Color.Transparent – key color az átlátszósághoz
- Graphics osztály
    - Graphics.Draw(Rectangle|Line)
    - Graphics.FillRectangle

- Pen
  - vastagság, szín, stílus (dotted)
- Brush
  - nem olyan mint a Pen, sokkal inkább egy tapéta;
  - kitöltési színt, mintát határoz meg
  - SolidBrush, HatchBrush, TextureBrush, LinearGradientBrush
  - előre definiált, pl. Brushes.Blue
- Bitmap
  - Image osztály áll egy Bitmap-ból és egy Metafile-ból
- az eddig említett eszközök (Graphics, Pen, ...) mind nagyon sokat esznek, ezért azonnal fel kell szabadítani őket (using, Dispose)
- egyéb képességek
  - antialiasing
  - jpg, png, gif, bmp támogatás
  - kép effektelés
  - unicode
- egyéb megjegyzések
  - ha lassú a rajzolás, dupla bufferelés: memóriában rajzolunk és csak a kész képet rakjuk ki a képernyőre

Generikus Típusok

C++

- C++-ban tanultuk, hogy kell, és hogy mik a jellemzői

.NET 1.0

- nincs generikus típus, vezess mindent vissza az ős-osztályra (object)
- macerás megoldás (castolni kell, szintén futási időben derül ki a hiba, keveredhetnek az objektumok)
- megoldás: leszármaztatni minden típus esetén egy specializált tárolót

.NET 2.0<

- generikus típusok (képességek: lásd Java)
- C++-szal ellentétben: lefordul IL kóddá a template is, majd a behelyettesítés is
  - a template hibái már fordítási időben kiderülnek
- referencia típusok esetén futásidőben helyettesítődik be, csak hivatkozások készülnek az objektumokra ezért nincs kódburjánzás mint C++-ban
- érték típusnál viszont minden template be lesz helyettesítve!
- reflexiót támogatja
- template változó helyén állhat: class, struct, interface, delegate, függvény
- implicit template példányosítás: (ha a függvény paraméterek között van template változó)
- kényszerek
  - struct: csak érték típus lehet (int, double, stb.)
  - class: csak referencia típusok lehetnek (osztályok)
  - konkrét osztálynév / interfész név: ez legyen az őse
  - new(): nem lehet letiltva a default konstruktor
  - pl.
```
class Name<T> where T : new() { ... }
class Name<T, U> where T : struct where U : class, new() { ... }
```

Java

- generikusság olyan mint a C#-ban
- template nem működik értékkel, csak referencia típussal
- kényszerek megadása szintén működik:
  ```
  public class Name<T extends Number & Comparable> { ... }
  ```
- viszont nem tudja: T = new T(), vagy T[] = new T[100];
- nem támogatja a referencia szerint átvételt
- viszont tud wildcard-ot: Collection<? extends Osztálynév>

Multithreading

Bevezetés

- egy program alatt:
  - több process, egy process alatt:
    - több thread

Process

- saját címtartomány
- rendszererőforrások
- legalább 1 thread (ha nem indítasz külön thread-et, az az 1 thread amiben vagy)
- def: védelmi egység – el vannak egymástól szigetelve, nincs hatással más process-re
- minden process adott egységnyi (sokszor nagyobbb mint a ram) virtuális címtartományt kap (szaron tanultuk)
- WinAPI: CreateProcess fgv. elindít egy folyamatot
- .NET: System.Diagnostics.Process.Start

Thread

- a thread egy process-en belül egy feladatot hajt általában végre
- benne vagyunk a main thread-ben, pl. a main függvény, ha indítunk thread-eket, multithread programunk lesz
- az OS ütemezi őket
- egy cpu core-on egyszerre egy thread futhat, (intelen 2, mert hyperthreading)
- állapotok
  - futó
  - felfüggesztett – OS felfüggesztette mert másik thread hajt végre valamit
  - I/O műveletre vár
- GUI alkalmazásoknál muszáj több szál, különben amíg pl. egy komponens animálódik, nem tudnánk kattintani
- szerver esetén is több szál kell, pl. minden kliens lekéréséhez (átlagosan jobb kihasználás)

Thread vs. Process

- a thread kis erőforrásigényű, könnyen kommunikálnak egymással
- minden thread-nek saját stack-je van, lokális változók rejtve
- a process-ek elszigeteltek, nehéz köztük kommunikálni

Szinkron/Aszinkron

- szinkron végrehajtás: a hívó a művelet befejezéséig várakozik

- aszinkron: a hívó nem várja meg a művelet befejezését, majd később valamilyen módon értesül az eredményről
- aszinkront hogyan?
    - OS belső megszakításával
    - külön szálban futtatással

Win32 szálkezelés

- CreateThread – elindít egy thread-et

.NET

- System.Threading névtérben keresgéljünk
- Thread osztály kezeli a thread-et pl.: `Thread t = new Thread(Function);`
- Start() metódus indítja a thread-et (paraméterezve is megadható a függvényünk)
- foreground/background-thread: a background futásának végét nem várja meg a fő szál (magyarul bezáródik a program), a foreground-ét igen
- Thread.GetCurrent() – akutális szálat adja vissza
- Thread.Name – nevet adhatunk a szálnak
- Thread.Sleep – CPU terhelése nélkül szüneteltetjük a szál futását
- szálak prioritása
    - effektív prioritás: priority(process)*priority(thread)
    - process prioritásai: Idle, BelowNormal, Normal, AboveNormal, RealTime
        - RealTime: kizárhatjuk az OS-t
    - thread prioritásai: Lowest, BelowNormal, Normal, AboveNormal, Highest
    - default: Normal
    - OS függő a pontos prioritás
    - Windows Forms: aktív ablak thread-nél javasolt a magasabb prioritást
        - kivéve, ha a process vezérlése sokba kerül, ilyenkor alacsonyabbat adjunk a GUI-nak

Realtime OS (kitérő)

- hard realtime
    - garantált a válaszidő egy előre megadott időn belül (repülőgép)
- soft relatime
    - kábé biztosított a válaszidő (Skype)
- automatikus szemétgyűjtés bezavar, kevésbé determinisztikus vele a program

Exception handling thread-ben

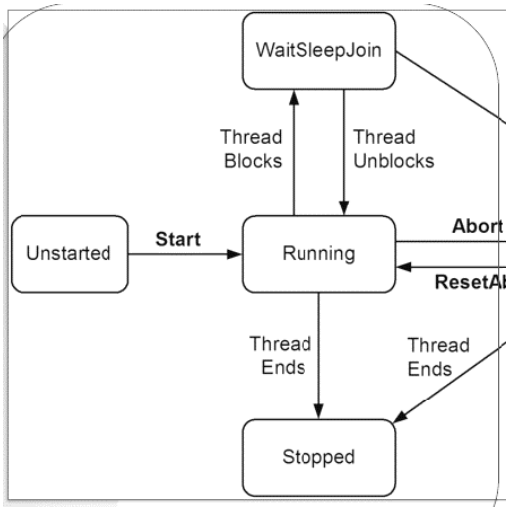
- nem jut ki a thread-ből az exception
- ThreadException kapja el őket

Thread-ek szinkronizációja

- mutual exclusion problémája
    - bizonyos erőforrásokhoz több szál fér hozzá, egyszerre nem írhatnak bele
    - be kell vezetni egy jelzést ami alapján tudjuk, hogy szabad-e az erőforrás
    - ez a jelzés egy atomi szintű művelet kell, hogy legyen, hogy ne lehessen pont rosszkor/túl későn megnézni az értékét
    - kritikus szakasz: ez a problémás erőforrással való interakció
- .NET

    - lock: C# utasítás, thread-ek közül mindig csak egy férhet hozzá
    - Mutex: processekre is tekintettel van, azokból is csak egy férhet hozzá
    - Semaphore: Mutex, de több process-t/thread-et engedélyez
    - ReaderWriterLock: sok olvasó, egy író
    - lock pont úgy működik mint a Javás synchronized (obj) { ... }
    - minden referencia típus lockolható
    - szálbiztos osztályok
        - csak amit muszáj, lassú a sok lockolás
        - C# - dokumentációban jelzi
        - CRL – van szálbiztos változata is
- Egyéb
    - 32 bites vagy annál kisebb típusokat nem kell lockolni, hiszen egy utasítás
    - C# - Interlocked osztály, atomi szintű aritmetikai utasítások nagyobb típusokra (double, long)
    - ne cache-elődjön egy változó a regiszterben: volatile (változó sorrend megmarad!)
    - szál kiléptetése: minden thread nézi, hogy bool end == true-e, ekkor kilép
- .NET szinkronizációs konstrukciók
    - lock megoldja a mutex problémát
    - jelzésre alkalmas függvények
        - EventWaitHandle – thread jelzésre vár
        - Monitor.Wait / Pulse – blokkolásra vár(?!)
    - WaitHandle
        - olyan osztályok őse amelyekre várakozni lehet
        - AutoResetEvent és ManualResetEvent eseményekre való várakozást tesz lehetővé
    - AutoResetEvent
        - .Set() – jelez az event, az egyik thread-ben lévő lock feloldódik (Java: notify())
        - .WaitOne() – ha az event jelzett, tovább fut (Java: wait())
        - .Reset() – nem jelzett állapotba állítja az event objektumot (?)
    - ManualResetEvent
        - Set() – ha jelzett, mindenki mehet (Java: notifyAll())
        - ha nem jelzett, mindenki vár
    - Mutex
        - lassabb mint a lock
        - WaitOne vár, ReleaseMutex elenged
    - Semaphore
        - lásd Mutex, de több process/thread férhet hozzá egyszerre
    - Thread.Interrupt használata
        - probléma: ha az osztályunk más osztályok metódusait hívja és ezekben blokkolás történik, nem tudunk flaget ellenőrizni
        - megoldás: Thread.Interrupt-ot meghívjuk a thread-en, és az dob egy exception-t, ha várakozik, így ki tudunk lépni a „távoli” loop-ból
    - Thread.Abort használata
        - olyan mint a Thread.Interrupt, de mindenhonnan kilépteti, még egy futó while ciklusból is
        - ne használjuk, csak ha a programból lépünk ki

- Szál állapotok:



  - WaitSleepJoin állapot (azt írja fontos...)
    - Sleep, Join, lock, WaitOne, stb. állapotok hatására kerül ide
    - hogy léphet ki belőle:
      - várakozás feltétel teljesül
      - timeout lejár
      - Thread.Interrupt/Abort hívására megszűnik a várakozás

Windows Forms Multithreading

- Probléma: Control objektumok csak a main thread-ből érhetőek el
  Mi van, ha egy másik thread-ből szeretnénk módosítani egy TextBox tartalmát?
  - Control.Invoke paranccsal beadhatunk egy függvényt a Control száljának
- Thread-pool
  - probléma: sok szált akarunk futtatni, viszont ez költséges és nem optimális
  - megoldás: program indulásakor indítunk thread-eket, ezekből allokálunk ha szükség van rá, ha befejeződik, visszakerül a pool-ba és újabb kérést szolgálhat ki
  - ha túl sok szál van, blokkoljuk a hívást, csak akkor indul el, ha van szabad szál

Threading összefoglalás

- növeli az alkalmazás komplexitását
  - szinkronizáció, mutex
- nehezen kinyomozható problémák
- túl sok szál terheli a rendszert
- deadlock alakulhat ki
  - több szál egymásra vár a lock miatt
  - timeout alkalmazása megoldást jelenthet

Bináris komponenesek, reflexió

Fordítás folyamata C/C++-ban

- prog2
- dll probléma: ha átírjuk a dll-t, a program elszáll
- megoldás: interfészen keresztül használjuk az osztályt, osztályokat
- ezt hívják COM-nak (Component Object Model)

.NET

- mi van a dll-ekkel? módosíthatom? igen, megoldották.

Reflexió

- lekérdezhetjük, hogy egy assembly-ben milyen típusok vannak
- azon belül függvényneveket, tagváltozókat, stb.
- pl. new Object().GetType
    - FieldInfo típussal tér vissza:
        - Name – tagváltozó neve
        - DeclaringType
        - IsPublic
        - MemberType – tagváltozó típusa

Saját attribútum

- System.Attribute osztályból származunk le
- [AttributeUsage(...)] határozza meg, hogy mire használjuk a tagváltozóinkat
- példa kód:

```
public static void Save(object o){
 Type type = o.GetType();
 object[] attributes = type.GetCustomAttributes(typeof(MyAttribute), false);
 if (attributes.Length == 0) return;
 FieldInfo[] fieldInfos = type.GetFields(BindingFlags.Public|...);
 foreach (FieldInfo fi in fieldInfos){
  object[] fA = fi.GetCustomAttributes(Type.GetType("StorableAttribute"), false)
  if (fA.Length == 0) continue;
  MyAttribute attr = (MyAttribute) fA [0];
  Console.WriteLine("Name: {0}", attr.Name);
  Console.WriteLine("Value: {0}", fi.GetValue(o).ToString());
 }
}
```

Adatbázis

Mi az adatbázis?

- jövő félévben tanultuk

SQL bevezető

- DDL (Data Definition Language) – tábla létrehozása, szerkezetének módosítása
- DML (... Manipulation ...) – rekordok beszúrása, módosítása, törlése, lekérdezése
- majd egyszer szoftlab5-ből már tanultuk

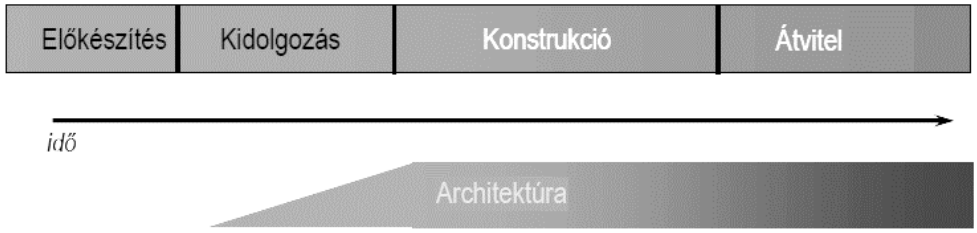
ADO.NET

- .NET környezetben adatbázis elérés
- SQL parancsok futtatása, eredmény elérése
- alternatívák
    - LINQ to SQL
    - Entity Framework
- Connection – kapcsolat a db felé
- Command – SQL parancsok futtatását teszi lehetővé
- DataReader – parancs utáni eredményhalmaz
- IConnection, DBConnection, Icommand, DbCommand, IDataReader, DbDataReader
    - interfészek és ősosztályok ADO.NET-hez

- Előző interfészek implementációi
    - Oracle: OracleConnection, ...
    - MSSQL: SqlConnection, ...
- Command műveletek
    - ExecuteNonQuery – nem tölti meg DataReader-t (INSERT, UPDATE)
    - ExecuteScalar – egy számmal tér vissza
    - ExecuteReader – egy vagy több rekorddal tér vissza (default?)
- DataReader-rel nem lehet többször végigmenni az eredményen (mint a C++ forward iterator)
- DataReader példány["oszlopnév"] visszaadja az aktuális sor oszlopának a tartalmát
- mivel költséges erőforrás az adatbázis, zárjuk le minél hamarabb
- DataSet alapú kapcsolat nélküli adathozzáférés
    - query után betöltődik a DataSet-ben az eredmény
    - kapcsolatot bontunk
    - DataSet-tel való interakció, mintha élő db-vel dolgoznánk
    - újra kapcsolódunk az adatbázishoz
    - DataSet változásait áttöltjük a db-be

Architekturális tervezés

Architektúra

- szoftverrendszer tervezése, strukturálása
- strukturális elemek kompozíciója, együttműködése
- architektúra-centrikusság
    - a modellek láthatóvá teszik, specifikálják, megalkotják és dokumentálják az architektúrát
    - unified process – futtatható architektúra sorozatos finomítását írja elő
    - 

- modell-nézet architekturális információtartalma
    - nem minden terv/modell architektúra
    - architekturális elem meghatározó a rendszer felépítése, teljesítménye szempontjából
        - amiből már nem lehet elvenni ahhoz, hogy megértsük és elmagyarázzuk a rendszer működését
- architektúra nézetei (elrendezés: csomag, alrendszer, dinamika: kölcsönhatás, állapotok)
    - use-case nézet – use-case-ek
    - tervezési nézet – osztályok, interfészek együttműködések
    - implementációs nézet – komponensek
    - process részek – aktív osztályok
    - telepítési nézet – csomópontok
- mi az architektúra? (különböző szempontból)
    - használati eset: funkcionális követelmények
    - tervezési vagy logikai: főbb csomagok, alrendszerek, osztályok
    - implementációs nézet: komponensek, dll-ek, exék, forráskódok szervezése

    - processz nézet: konkurrens aspektus, folyamatok, szálak, deadlock, teljesítmény
    - telepítési nézet: futtatható és egyéb komponensek mely számítási csomópontokra kerülnek
- architektúra és funkcionalitás
    - az architektúrának és a funkcionalitásnak egyensúlyban kell lenni
    - az architektúra legyen jól strukturált, a funkció működjön jól
    - klasszikus rendszer: az architektúra forrása nagyrészt az ún. lopás, és másodlagosan a módszer és intuíció
    - ritka rendszer: leginkább intuíció
    - mi a lopás? jól bevált megoldások átvétele

Rétegelés

- alapvető szervezési elv: kódot rátegekbe szervezzük
- előfordulása
    - OS felépítésénél – driver – OS – alkalmazás
    - hálózat
    - vállalati információs rendszerek
- i-edik réteg szolgáltatást nyújt az i+1-edik rétegnek
- rétegeken belül tetszőleges függőség lehet a komponensek között
    - egymástól független komponenseket egy rétegen belül csoportosítjuk
    - a részek cserélhetők, fejleszthetők
- rétegezési példa: driverek-nél
    - driver, alatta polling
    - driver, alatta interrupt
- rétegelés előnye
    - ha a feladat túl komplex, vezessünk be absztrakciós szinteket
    - a rétegek használatának megértéséhez nem kell a többi réteget ismerni
    - többféleképpen használhatóak föl (hálózati protokollok)
- rétegelés hátránya
    - egyszerű feladatnál felesleges
    - a nagyobb változások végigvonulnak az összes rétegen
- Layer
    - logikai réteg, ezt tervezzük meg először, eddig erről volt szó
    - célszerű külön modulba tenni
- Tier
    - fizikai réteg, más gépen fut, hálózati kommunikáció jellemző rá
    - a logikai rétegekre mondjuk meg mely gépeken fussanak

Információs rendszerek

- nincsenek rétegek: batch rendszer
- kétrétegű architektúra: kliens – szerver
- háromrétegű architektúra
- SOA architektúra

Kétrétegű architektúra

- az adat megosztott, a feldolgozás/megjelenítés elosztott
- pl. megosztott adatbázis
- alkalmazások

- ún. kliens-szerver működés
- előnyei
    - adatkarbantartás elkülöníthető az alkalmazási rétegtől
    - több alkalmazással is hozzáférhetünk a kliensen a szerverhez
    - RAD támogatás miatt népszerű
- hátrányai
    - validálás hova kerül? adatintegritás, üzleti szabályok, számítások?
    - ha kliens oldalon van, több helyen kell karban tartani
    - keveredik az üzleti logika a GUI-val
    - adatbázisban nem nagyon támogatott a validálás és nem is ez a dolga
- mikor használjuk
    - egyszerű alkalmazásoknál

Háromrétegű architektúra

- rétegek
    - alkalmazás (pl. UI)
    - tartomány – üzleti logika
    - adatbázis – belső sémával
- előnyök
    - üzleti logika nincs duplikálva az alkalmazásban
    - könnyen újrafelhasználható (webes és vastag kliens)
    - az alkalmazás kevésbé függ a fizikai adatszerkezettől és az adatbázis helyétől
    - db átszervezhető az alkalmazástól függetlenül
- hátrányok
    - extra komplexitás
    - nagy erőforrásigény
    - üzleti logikai komponenseket le kell képezni relációs modellre, ami nehéz

Microsoft háromrétegű modellje

- nem csak microsoft, minden környezetben ezek szoktak megjelenni
- adathozzáférési logika (Data Access Layer): elrejti az adatkezelő rendszer
- szolgáltatásügynökök: külső rendszerrel való kommunikáció
- üzleti entitás: adatokat reprezentálnak (megrendelés, vásárló)
- üzleti komponensek: üzleti logikát valósítják meg (pl. megrendelés felvétele, listázás, üzleti számítások, stb.)
- üzleti munkafolyamatok: bonyolultabb üzleti folyamatok esetén lépéseket definiálnak, a folyamat levezénylése a feladatuk
- szolgáltatásinterfészek: vékony réteg; kliensek csak ezen keresztül férhetnek hozzá az üzleti szolgáltatásokhoz, belső komponensek szabadon átszervezhetőek
- felhasználói felület komponensei: felhasználói felület űrlapjai, ablakai rajtuk lévő logikával
- felhasználói folyamat komponensek: bonyolultabb felhasználói folyamatok interakciók levezénylése (kisebb jelentőségű, ritkán használt)

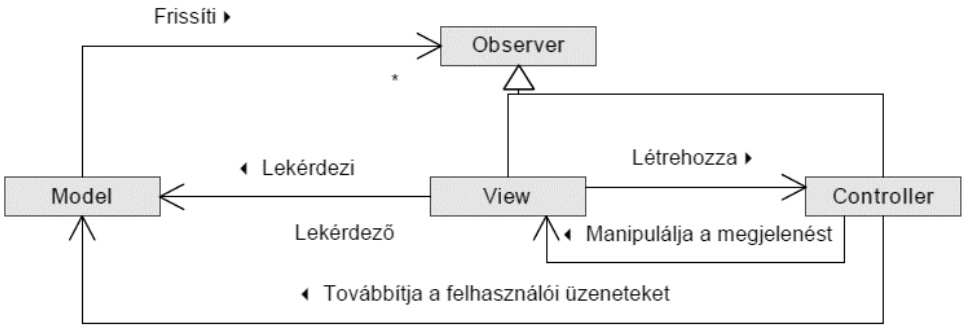
Document-View architektúra

- leginkább Excel-szerű programoknál
- ne keverjük a GUI-ba az alkalmazáslogikát
- válasszuk ketté az adatok kezelését és a megjelenítést
- másik megoldás az MVC

- felépítés:
  - Document
    - feladata az adatok tárolása, menedzselése, modellnek is nevezik
    - olyan osztályok melyek adatokat tagváltozóikban tárolják és olyan tagfügvényekkel rendelkeznek melyek kezelik ezeket (Load, Save, stb.) elérhetővé teszik ezt más osztályok számára (View részére)
  - View
    - feladata az adatok megjelenítése és az interakciók kezelése
    - ez általában az ablak vagy egy tab
- több nézet esetén hogy frissítsünk?
  - kezeljük az adatokat külön dokumentumként, a modellre view-kat regisztrálunk be
  - ha valamelyik view megváltoztatja a dokumentum adatait, értesíti a többit a változásról

Model-View-Controller architektúra

- leginkább GUI-s programoknál
- ne legyen köze az alkalmazásnak a megjelenítéshez
- felépítés
  - Model – alkalmazáslogika
    - tartalmazza az adatokat valamint a műveleteket
  - View – megjelenítés
    - megjeleníti az adatokat amit a modellből olvas ki
  - Controller – interakció
    - a felhasználói interakciót kezeli
  - Observer
    - ebben tárolódnak a View-k és Controller-ek, garantálja hogy konzisztensen mutassa az adatokat a View és a Controller a Model-nek megfelelően viselkedjen



- előnyök
  - többféle nézet ugyanannak az adatnak
  - szinkronizált nézetek
  - függetlenül cserélhető bővíthető view-k és controllerek
  - a modell könnyebben tesztelhető
  - lehetséges framework(???)
- hátrányok
  - komplexitás növekszik
  - túl sok szükségtelen frissítés
  - view és controller gyakran nem különválasztható vagy nem is célszerű

        - MVC csak a webes világban elterjedt

Piping

- adatfolyamot feldolgozó rendszer
- szűrőkből áll
- csővezetékek kötik össze
- többféleképpen kombinálható
- pl. videó feldolgozás, banki jelentések
- részei: forrás, nyelő, szűrők, vezetékek
- szűrők
    - transzformációt hajtanak végre
    - aktív: behív, kiad (szinkronizálhatóak)
    - passzív: megkap, elveszik tőle
- pl. FIFO
- hiba esetén kérdéses a folytatás