2009. április
Halacs

Név: ..... **Neptun kód:** .....

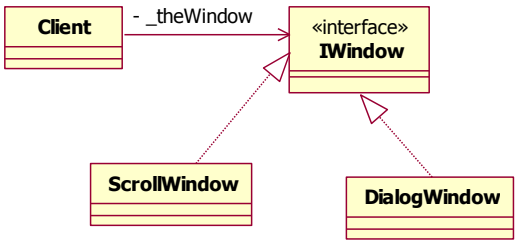
Szoftvertechnikák BSc zárthelyi dolgozat 2007.04.13

Ponthatárok	
0-49	Elégtelen (1)
50-60	Elégséges (2)
61-70	Közepes (3)
71-84	Jó (4)
85-	jeles(5)

Pontok		
kérdés	max	pont
1	15	
2	15	
3	15	
4	10	
5	15	
6	15	
7	15	
∑	100	
Jegy		

Jegy:

1) Modellezés
   a) Adja meg az alábbi osztálydiagramnak megfelelő C#, Java **vagy** C++ kódot. Csak a *Client* és *IWindow* elemekre térjen ki! (8p)




```
class Client
{
      private IWindow theWindow;

}

interface IWindow
{
}

class ScrollWindow : IWindow
{
}

class DialogWindow : IWindow
{
}
```

   b) C# nyelvű környezetben egy `List<Shape> shapes;` listában különböző *Shape* leszármazott osztálybeli objektumokat tárolunk. A *Shape* egy ***absztrakt osztály***. A fejlesztés során a feladatunk az, hogy egy általunk megírandó *Rectangle* osztály objektumait is tudjuk a *shapes* listában tárolni azzal a feltétellel, hogy a *Rectangle* osztálynak kötelezően kell legyen egy *RectBase* ősosztálya, ami nem a *Shape*-ből származik és aminek a forráskódjához nem férhetünk hozzá.
   A feladat: Szövegesen adja meg, hogy hogyan kell a megoldásunkat megváltoztatni ahhoz, hogy a *Rectangle* osztály objektumai is betehetőek legyenek a *shapes* listába. A *RectBase* osztály nem megváltoztatható! (7p)

   A Shape absztakt osztályból interfészt kell csinálni, az esetleges adattagjait és fv-eit pedig a származtatott osztályokban kell megvalósítani.

2) Ismertesse röviden a .NET property (tulajdonság) fogalmát és néhány jellemző felhasználási esetét. Mutasson rövid kódrészletet definiálására és használatára. (15p)

A tulajdonságok segítségével osztályok tagváltozóihoz férhetünk hozzá szintaktikailag hasonló módon, mintha egy hagyományos tagváltozót érnénk el. A hozzáférés során azonban lehetőségünk van, hogy az egyszerű érték lekérdezés vagy beállítás helyett metódusszerűen implementáljuk a változó elérésének a módját.

Lehetőség van a hozzáférést korlátozni (csak olvasható), illetve íráskor ellenőrzést végezni az új adaton (nem biztos, hogy szerencsés, ha betudunk állítani irreális születési évet).

Csak olvasható vagy csak írható property esetén fordítási időben hibát kapunk, ha nem engedélyezett művelettel próbálkozunk.

Jellemző használatra példa: irreális születési év beállításának elkerülése, számított mező létrehozása.

```
class Ember
{
    private int szuletesi_ev;           // ezt fogjuk a propertyvel piszkálni

    public int Szuletesi_Ev
    {
        set                             // beállító "üggvény"
        {
            if (value > 1900)           // Ellenőrzöm az új értéket jó-e
                szuletesi_ev = value;
            else
                throw new ArgumentException("Rossz évet adtál meg!");
        }

        get                             // lekérdező "függvény"
        {
            return szuletesi_ev;
        }
    }

    public int kor                      // ez egy csak olvasható property
    {
        get
        {
            return DateTime.Now.Year - szuletesi_ev;
        }
    }
}
```

3) Eseményvezérelt programozás és grafikus megjelenítés.
   a) Ismertesse röviden az érvénytelen terület fogalmát! Hogyan kapcsolódik ez a Paint eseményhez? (7p)

Korábban takarásban levő, láthatóvá vált ablakrészek (pl. átméretezés, Z-orderben előbb került az ablak, stb.)

Érvénytelen terület keletkezésekor meghívódik a Paint függvény, mely gondoskodik a terület (ablak) újrarajzolásáról.

   b) Írjon olyan C# nyelvű alkalmazásrészletet, ami a (10,10) koordinátában megjeleníti az ablakfrissítések (paint) számát! Csak a számláláshoz és megjelenítéshez kapcsolódó kódrészeket adja meg. (8p)

Lényegében a kapott WM_PAINT-ek számát kell kiírni! Ilyet akkor kapunk, ha érvénytelen terület keletkezett, vagyis ha meghívódik az OnPaint kirajzoló függvény.

```
int counter = 0;

protected override void OnPaint(PaintEventArgs e)
{
      base.OnPaint(e);
      counter++;
      e.Graphics.DrawString(counter.ToString(), this.Font,
                                                Brushes.Black, 10, 10);
}
```

Az átméretezés alapértelmezésben nem festi újra magát, engedélyezni kell (a form/control stílusát kell állítani, **pl. a konstruktorban**):

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

Ekkor meg villog átméretezéskor. Engedélyezzük a duplapufferelést:

```
SetStyle(ControlStyles.DoubleBuffer | ControlStyles.UserPaint |
                              ControlStyles.AllPaintingInWmPaint, true);
UpdateStyles();
```

4) A destruktor és a dispose tervezési minta felügyelt .NET környezetben

a) Milyen esetben célszerű destruktort írni C# nyelven? (3p)

Csak ha „drága”, nem felügyelt erőforrást foglal az objektumunk (pl. natív ablak leíró, adatbáziskapcsolat, file, lock, stb). Egyébként ne írjunk, mert a destruktorral rendelkező osztályok megszűnés előtt bekerülnek a finalizer queue-ba: lassít!

(C#-ban a destruktor a Finalize függvény formájában van jelen, ellentétben a C/C++-al, ahol nyelvi szinten támogatott)

b) Mik a destruktor hívásának jellemzői? (3p)

- A végrehajtás ideje nem ismert!
  Vagy a GC (szemétgyűjtő) szabadít fel, vagy explicit függvényhívással (ha szükséges)
- A sorrend nem ismert!
  Ha pl. 100 objektum szabadítható fel, milyen sorrendben szabadulnak fel (ezek hivatkozhatnak is egymásra!)...
- A szál nem ismert!
- A destruktor csak a külső hivatkozásait engedheti el!
  (pl. nem felügyelt erőforrások, mert a felügyelteket már lehet felszabadította a szemétgyűjtő)

c) A b) kérdéshez kapcsolódóan: milyen célt szolgál a *dispose* tervezési minta? Röviden ismertesse a megoldás alapelvét (részletes kód nem szükséges)! (4p)

A dispose tervezési mintának köszönhetően az objektumok által foglalt erőforrások felszabadítása válik determinisztikussá.

Implementáljuk az IDisposable interfészt → Minden osztály tartalmazzon Dispose() függvényt, melynek törzsében felszabadítjuk az erőforrásokat. Ez explicit hívható és hívandó függvény lesz. A tartalmazott, IDisposable-t implementáló felügyelt objektumokra is hívjunk Dispose-t, hogy a tartalmazott objektumok is fel tudják szabadítani a nem felügyelt erőforrásaikat.
A destruktorban is gondoskodjuk a nem felügyelt erőforrások felszabadításáról, hogy akkor se szivárogjanak el, ha a fejlesztő elfelejtett Dispose-t hívni.

5) Szálkezelés

a) Egy mondatban adja meg, miért könnyebb az egy folyamaton belüli szálak kommunikációját megvalósítani, mint a folyamatok közöttit! (3p)

Az egy folyamaton belüli szálak azonos memóriaterületet használnak, így a szálak elérik egymás változóit.

b) Adja meg röviden a többszálú alkalmazások három előnyét! (4p)

1. jobb processzor kihasználtság
2. nem növekvő átlagos válaszidő (interaktivitás – pl. nem akad le a GUI, szerveralkalmazások)
3. időzítés érzékeny feladatok magasabb prioritású szálon futtathatóak

c) Adja meg a szálbiztos (thread-safe) osztály fogalmát egy mondatban! (3p)

Szálbiztos osztályoknak nevezzük azokat az osztályokat, melyeknek nyilvános interfésze adatintegritás sérüléssel szemben védett, így többszálú környezetben is biztonságosan használhatók.

Vagyis: ha több szálból is használjuk ugyanazt az osztálypéldányt, akkor előfordulhat, hogy olyan szerencsétlenül történnek a függvényhívások és az átütemezés az egyes szálak között, hogy ez hibás állapotot, végül hibás működést eredményez. A szálbiztos osztályok úgy vannak elkészítve, hogy ilyen nem fordulhat elő. (pl. kiritikus régiók)

d) Egészítse ki az alábbi C# osztályt úgy, hogy szálbiztos legyen egy adott folyamaton belül és röviden indokolja megoldását! A megoldást a feladatlapon adja meg! (5p)

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

  public void Push(T item) {

        lock (syncroot)
        {
          items[current++] = item;
        }

  }

  public T Pop() {

        lock (syncroot)
        {
          return items[current--];
        }
  }
}
```

A push és pop metódusok kritikus régiókba kerültek, mivel itt változik az osztály belső állapota olyan módon, hogy ha közbe átütemezés következne be, majd másik szálból újrahívódna, inkonzisztens állapot alakulhatna ki. Inkonzisztens állapot pontosabban akkor következhet be, ha a current változó már módosítva van, de még nem lett eltárolva/visszaadva a hozzá tartozó adat. Ilyenkor egy másik szál elronthatja a current változó értékét, amivel rossz indexelést idéz elő így végső soron nem a kívánt címre történik a hivatkozás.

6) Definiálja és hasonlítsa össze a statikus és a dinamikus programkönyvtárakat! (15p)

A statikus programkönyvtárak fordításkor linkelődnek hozzá a programhoz, így a program indulásától kezdve a memóriában tárolódnak. Ebből adódóan annyiszor töltődik be, ahány példányban fut a program.

A dinamikus programkönyvtárak (DLL fájlok – Dinamic Load Library) futásidőben, igény szerint töltődnek be a memóriába. Fordításkor, csak a hivatkozások kerülnek tárolásra. Csak egy példányban töltődik be, akárhány program használja.

7) Ismertesse egy rövid C# nyelvű példán keresztül az ADO.NET kapcsolatalapú adathozzáférését! (15p)

Ez most nem fog kelleni az volt a 2009-04-01 előadáson.