Eötvös Loránd Tudományegyetem
Informatikai Kar

Eseményvezérelt alkalmazások fejlesztése II

4. előadás

Windows Forms alkalmazások architektúrája és tesztelése

Giachetta Roberto

groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

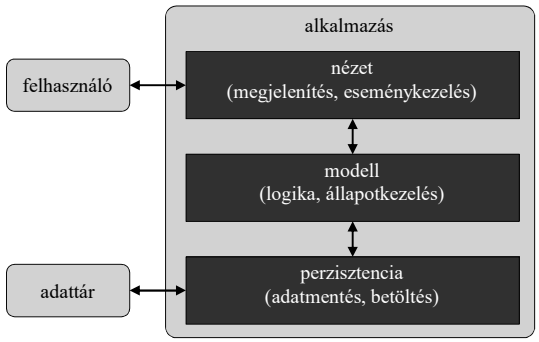
Windows Forms alkalmazások architektúrája

Alkalmazások architektúrája

- *Szoftver architektúrá*nak nevezzük a szoftver fejlesztése során meghozott *elsődleges tervezési döntések* halmazát
  - célja *a rendszer magas szintű felépítésének és működésének meghatározása*, a komponensek és relációk kiépítése
  - a tervezés során általában mintákra hagyatkozunk, ezeket nevezzük *architekturális mintá*knak (*architectural pattern*)
- A *háromrétegű* (*three-tier*) architektúra a leggyakrabban alkalmazott szerkezeti felépítés, amelyben elkülönül:
  - a *nézet* (*presentation/view tier*, *presentation layer*)
  - a *modell* (*logic/application tier*, *business logic layer*)
  - a *perzisztencia*, vagy *adatelérés* (*data tier*, *data access layer*, *persistence layer*)

Windows Forms alkalmazások architektúrája

A háromrétegű architektúra



Windows Forms alkalmazások architektúrája

Függőségek

- Az egyes rétegek között *függőség*ek (*dependency*) alakulnak ki, mivel felhasználják egymás funkcionalitását
  - a cél a minél kisebb függőség elérése (*loose coupling*)
  - ezért a függőségeket úgy kell megvalósítanunk, hogy a konkrét megvalósítástól ne, csak annak felületétől (interfészétől) függjünk
- A rétegek a függőségeknek csak az absztrakcióját látják, a konkrét megvalósítást külön adjuk át nekik, ezt nevezzük *függőség befecskendezés*nek (*dependency injection*)
  - a befecskendezés helye/módszere függvényében lehetnek különböző típusai (pl. konstruktor, metódus, interfész)

Windows Forms alkalmazások architektúrája

Függőségek

- Pl. :

```
interface IDependency // függőség interfésze
{
   Boolean Check(Double value);
   Double Compute();
}
…
class DependencyImplementation : IDependency
   // a függőség egy megvalósítása
{
   public Boolean Check(Double value) { … }
   public Double Compute() { … }
}
```


Windows Forms alkalmazások architektúrája

Függőségek

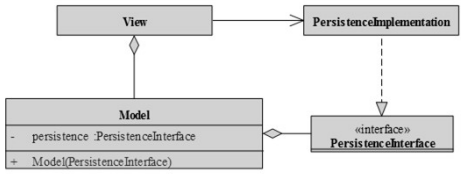
- Pl. :

```
class Dependant { // osztály függőséggel
   private IDependency _dependency;

   public Dependant(IDependency d) {
      _dependency = d;
   } // konstruktor befecskendezéssel helyezzük be
     // a függőséget
   …
}
…
Dependant d =
   new Dependant(new DependencyImplementation());
   // megadjuk a konkrét függőséget
```

Windows Forms alkalmazások architektúrája
Függőségek

- Háromrétegű architektúra esetén a függőség befecskendezést használhatjuk a modell, illetve az adatkezelés esetén is
  - pl. az adatkezelés esetén elválasztjuk a felületet (`PersistenceInterface`) a megvalósítástól (`PersistenceImplementation`), utóbbit a nézet fogja befecskendezni a modellbe



Windows Forms alkalmazások architektúrája
Fájlkezelés

- Az adatfolyamok kezelése egységes formátumban adott, így azonos módon kezelhetőek fájlok, hálózati adatforrások, memória, stb.
  - az adatfolyamok ősosztálya a `Stream`, amely binárisan írható/olvasható
- Szöveges adatfolyamok írását, olvasását a `StreamReader` és `StreamWriter` típusok biztosítják
  - létrehozáskor megadható az adatfolyam, vagy közvetlenül a fájlnév
  - csak karakterenként (`Read`), vagy soronként (`ReadLine`) tudunk olvasni, így konvertálnunk kell
  - amennyiben a műveletek során hiba keletkezik, `IOException`-t kapunk

Windows Forms alkalmazások architektúrája
Fájlkezelés

- Pl.:

```
try
{
   StreamReader reader =
      new StreamReader("in.txt"); // megnyitás
   while (!reader.EndOfStream) // amíg nincs vége
   {
      Int32 val = Int32.Parse(reader.ReadLine());
      // sorok olvasása, majd konvertálás
      …
   }
   reader.Close(); // bezárás
 }
 catch (IOException) { … }
```


Windows Forms alkalmazások architektúrája
Erőforrások felszabadítása

- A referencia szerinti változók törlését a szemétgyűjtő felügyeli
  - adott algoritmussal adott időközönként pásztázza a memóriát, törli a felszabadult objektumokat
  - sok, erőforrás-igényes objektum példányosítása esetén azonban nem mindig reagál időben, így nő a memóriahasználat
  - a `GC` osztály segítségével beavatkozhatunk a működésbe
- A manuális törlésre (destruktor futtatásra) nincs lehetőségünk felügyelt blokkban, de erőforrások felszabadítására igen, amennyiben az osztály megvalósítja az `IDisposable` interfészt, és benne a `Dispose()` metódust

Windows Forms alkalmazások architektúrája
Erőforrások felszabadítása

- Emellett a C# nyelv tartalmaz egy olyan blokk-kezelési technikát, amely garantálja a `Dispose()` automatikus futtatását:

```
using (<objektum példányosítása>)
{
   <objektum használata>
} // itt automatikusan meghívódik a Dispose()
```

- Pl.:

```
using (StreamReader reader = new StreamReader(…)){
   // a StreamReader is IDisposable
   …
}
// itt biztosan bezáródik a fájl, és
// felszabadulnak az erőforrások
```

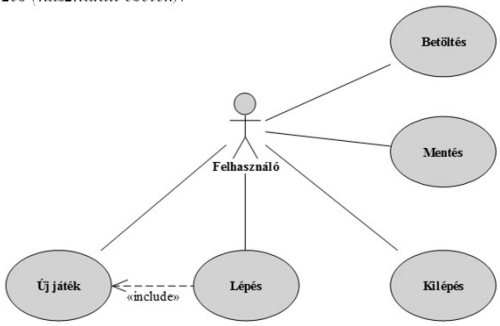

Windows Forms alkalmazások architektúrája
Példa

Feladat: Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- a programban lehetőséget adunk új játék kezdésére, valamint lépésre (felváltva)
- a programban ‚X' és ‚0' jelekkel ábrázoljuk a két játékost
- a program automatikusan jelez, ha vége a játéknak (előugró üzenetben), majd automatikusan új játékot kezd, és a játékos bármikor kezdhet új játékot (`Ctrl+N`)
- lehetőséget adunk játékállás elmentésére (`Ctrl+L`) és betöltésére (`Ctrl+S`), a fájlnevet a felhasználó adja meg
- a programot háromrétegű architektúrában valósítjuk meg

Windows Forms alkalmazások architektúrája
Példa

Tervezés (használati esetek):



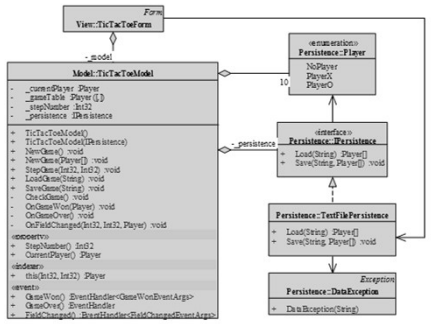
Windows Forms alkalmazások architektúrája
Példa

Tervezés (architektúra):

- létrehozunk egy adatelérési névteret (**`Persistence`**), ebben egy interfész (**`IPersistence`**) biztosítja a betöltés (**`Load`**) és mentés (**`Save`**) funkciókat
- az adatelérés egy tömböt (**`Player[]`**) használ a modellel történő kommunikációra, amely sorfolytonosan tartalmazza az értékeket
- megvalósítjuk az interfészt szöveges fájl alapú adatkezelésre (**`TextFilePersistence`**)
- a nézet befecskendezi a modellbe a fájl alapú adatkezelést, ami a betöltés (**`LoadGame`**) és mentés (**`SaveGame`**) műveleteivel bővül

Windows Forms alkalmazások architektúrája
Példa

Tervezés (szerkezet):



Windows Forms alkalmazások architektúrája
Példa

*Megvalósítás (**`TextFilePersistence.cs`**):*

```
public Player[] Load(String path) {
    if (path == null)
        throw new ArgumentNullException("path");

    try {
        using (StreamReader reader =
                new StreamReader(path))
                // fájl megnyitása olvasásra
        {
            String[] numbers =
                reader.ReadToEnd().Split();
                // fájl tartalmának feldarabolása a
                // whitespace karakterek mentén
```


Windows Forms alkalmazások architektúrája
Példa

*Megvalósítás (**`TextFilePersistence.cs`**):*

```
            // a szöveget számmá, majd játékossá
            // konvertáljuk, és ezzel a tömbbel
            // visszatérünk
            return numbers.Select(number =>
              (Player)Int32.Parse(number)).ToArray();
            ...
        } // bezárul a fájl
    }
    catch { // ha bármi hiba történt
        throw new TicTacToeDataException("Error
                          occured during reading.");
    }
}
```

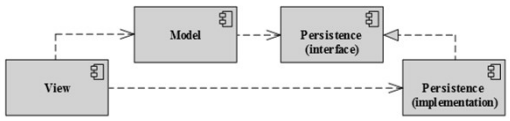

Windows Forms alkalmazások architektúrája
Szerelvények

- A szoftver egyes csomagjait fizikailag is elválaszthatjuk egymástól azáltal, hogy külön *szerelvények*be (*assembly*) helyezzük őket, ez által az alkalmazás komponenseivé válnak
  - a szerelvény típusok és erőforrások lefordított, felhasználható állománya, pl. az *alkalmazás* (*executable*, *.exe*)
  - az *osztálykönyvtárak* (*class library*, *.dll*) olyan szerelvények, amelyek önállóan nem futtathatóak, csupán osztályok gyűjteményei, amelyek más szerelvényekben felhasználhatóak
    - a nyelvi könyvtár is osztálykönyvtárakban helyezkedik el
- A Visual Studio-ban minden projekt egy külön szerelvényt eredményez, a megoldás (*Solution*) fogja össze az egy szoftverhez tartozó szerelvényeket

Windows Forms alkalmazások architektúrája

Felbontás szerelvényekre

- Az alkalmazások felbontása több szempontból is hasznos:
  - elősegíti az egyes programrészek szeparálását, a függőségek korlátozását, a komponensek újrahasznosítását
  - megkönnyíti a csapatmunka felosztását, a keletkezett kódok összeintegrálását, tesztelését, publikálását
- A felosztást legcélszerűbb a rétegek és függőség befecskendezés mentén elvégezni, pl.:



Windows Forms alkalmazások architektúrája

Példa

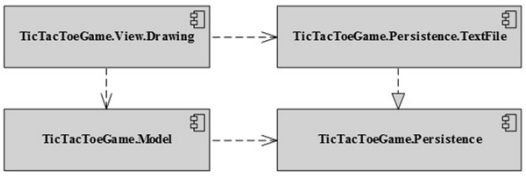
Feladat: Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- az alkalmazást háromrétegű architektúrában valósítjuk meg, az adatelérést befecskendezzük a modellbe
- emiatt négy projektbe szeparáljuk a forrást:
  - nézet (`TicTacToeGame.View.Drawing`)
  - modell (`TicTacToeGame.Model`)
  - adatkezelés felülete (`TicTacToeGame.Persistence`)
  - adatkezelés szöveges fájl alapú megvalósítása (`TicTacToeGame.Persistence.TextFile`)
- a nézet az alkalmazás, a többi projekt osztálykönyvtár

Windows Forms alkalmazások architektúrája

Példa

Tervezés (architektúra):



Windows Forms alkalmazások architektúrája

Példa

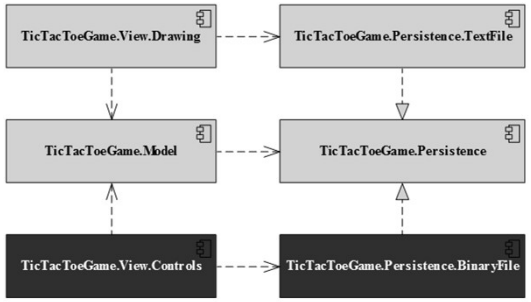
Feladat: Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- helyezzük vissza a korábbi, vezérlő alapú grafikus felületet a programba egy új alkalmazás projektben (`TicTacToeGame.View.Controls`)
- készítsünk egy új, bináris fájl alapú adatelérést (`TicTacToeGame.Persistence.BinaryFile`)
  - csupán az értékeket írjuk ki és olvassuk be bájtonként a `File` osztály `ReadAllBytes(…)` és `WriteAllBytes(…)` műveletei segítségével
  - használjuk az új típusú adatelérést az új nézetben

Windows Forms alkalmazások architektúrája

Példa

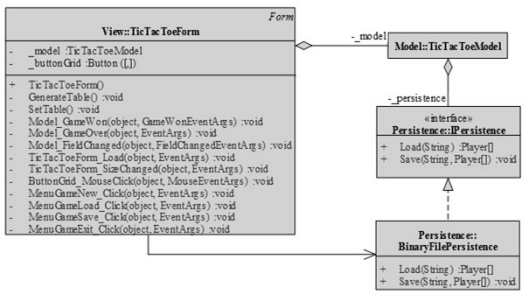
Tervezés (architektúra):



Windows Forms alkalmazások architektúrája

Példa

Tervezés (szerkezet):



Windows Forms alkalmazások architektúrája
Példa

Megvalósítás (`BinaryFilePersistence.cs`*):*

```
public Player[] Load(String path) {
   …
   try {
      Byte[] fileData = File.ReadAllBytes(path);
         // fájl bináris tartalmának beolvasása

      // konvertálás és tömbbé alakítás
      return fileData.Select(fileByte =>
                        (Player)fileByte).ToArray();
   }
   …
}
```


Windows Forms alkalmazások tesztelése
Tesztelés

- A programoknak minden esetben alapos tesztelésen kell átesnie
  - a dinamikus tesztelést a rendszer különböző szintjein végezzük (egységteszt, integrációs teszt, rendszerteszt)
- Az *egységteszt* (*unit test*) egy olyan automatikusan futtatható ellenőrzés, amely lehetőséget osztályok és objektumok viselkedésének ellenőrzésére (a tényleges viselkedés megegyezik-e az elvárttal)
  - a Visual Studio lehetőséget ad, hogy egységteszteket automatikusan generáljunk és futtassunk le
  - az egységtesztek külön projektbe kerülnek (*Unit Test Project*), amelyből meghivatkozzuk a tesztelendő projektet

Windows Forms alkalmazások tesztelése
Egységtesztek

- Az egységtesztek a megvalósításban osztályok a `TestClass` attribútummal jelölve
  - a tesztesetek eljárások (a `TestMethod` attribútummal jelölve), amelyeket automatikusan futtatunk
  - a tesztek az `Assert` osztály segítségével végeznek ellenőrzéseket (`AreEqual`, `IsNotNull`, `IsFalse`, `IsInstanceOfType`, …), és különböző eredményei lehetnek (`Fail`, `Inconclusive`)
  - lehetőségünk van a teszteket inicializálni (`TestInitialize`, `TestCleanup`)
  - a teszt rendelkezik egy környezettel (`TestContext`), amely segítségével lekérdezhetünk információkat

Windows Forms alkalmazások tesztelése
Egységtesztek

- Pl.:

```
[TestClass] // tesztosztály
public class RationalTest {
   …
   [TestMethod] // tesztművelet a konstruktorra
   public void RationalConstructorTest(){
      Rational actual = new Rational(10, 5);
      Rational target = new Rational(2, 1);
         // az egyszerűsítést teszteljük
      Assert.AreEqual(actual, target);
         // ha a kettő egyezik, akkor eredményes a
         // teszteset
   }
}
```


Windows Forms alkalmazások tesztelése
Példa

Feladat: Teszteljük a TicTacToe játékot.

- az egységtesztet egy új tesztprojektben (`TicTacToeGame.Test`) hozzuk létre, és meghivatkozzuk a modell projektet
- a tesztosztályban (`TicTacToeModelTest`) ellenőrizzük:
  - a konstruktor működését, és az üres tábla létrejöttét (`TicTacToeConstructorTest`)
  - léptetés értékbeállításait (`TicTacToeStepGameTest`)
  - lépésszám számlálást (`TicTacToeStepNumberTest`)
  - játék vége eseményét, és annak paraméterét (`TicTacToeGameWonTest`)

Windows Forms alkalmazások tesztelése
Példa

Megvalósítás (`TicTacToeModelTest.cs`*):*

```
[TestClass]
public class TicTacToeModelTest {
   // egységteszt osztály
   [TestMethod]
   public void TicTacToeConstructorTest() {
      // egységteszt művelet
      …
      for (Int32 i = 0; i < 3; i++)
          for (Int32 j = 0; j < 3; j++)
              Assert.AreEqual(Player.NoPlayer,
                              _model[i, j]);
              // valamennyi mező üres
      }
```

Windows Forms alkalmazások tesztelése
Mock objektumok

- Amennyiben függőséggel rendelkező programegységet tesztelünk, a függőséget helyettesítjük annak szimulációjával, amit *mock objektum*nak nevezünk
  - megvalósítja a függőség interfészét, egyszerű, hibamentes funkcionalitással
  - használatukkal a teszt valóban a megadott programegység funkcionalitását ellenőrzi, nem befolyásolja a függőségben felmerülő esetleges hiba
- Mock objektumokat manuálisan is létrehozhatunk, vagy használhatunk erre alkalmas programcsomagot
  - pl. *NSubstitute*, *Moq* letölthetőek NuGet segítségével

Windows Forms alkalmazások tesztelése
Mock objektumok

- Pl. :

```
class DependencyMock : IDependency
   // mock objektum
{
   // egy egyszerű viselkedést adunk meg
   public Double Compute() { return 1; }
   public Boolean Check(Double value) {
      return value >= 1 && value <= 10;
   }
}
…
Dependant d = new Dependant(new DependencyMock());
   // a mock objektumot fecskendezzük be a függő
   // osztálynak
```


Windows Forms alkalmazások tesztelése
Mock objektumok

- *Moq* segítségével könnyen tudunk interfészekből mock objektumokat előállítani
  - a `Mock` generikus osztály segítségével példányosíthatjuk a szimulációt, amely az `Object` tulajdonsággal érhető el, és alapértelmezett viselkedést produkál, pl.:

```
Mock<IDependancy> mock =
   new Mock<IDependancy>();
   // a függőség mock objektuma
Dependant d = new Dependant(mock.Object);
   // azonnal felhasználható
```

  - a `Setup` művelettel beállíthatjuk bármely tagjának viselkedését (`Returns(…)`, `Throws(…)`, `Callback(…)`), a paraméterek szabályozhatóak (`It`)

Windows Forms alkalmazások tesztelése
Mock objektumok

- pl. :

```
mock.Setup(obj => obj.Compute()).Returns(1);
   // megadjuk a viselkedést, mindig 1-t ad
   // vissza
mock.Setup(obj =>
           obj.Check(It.IsInRange<Double>(0, 10,
                     Range.Inclusive)))
    .Returns(true);
mock.Setup(obj => obj.Check(It.IsAny<Double>())
    .Returns(false);
    // több eset a paraméter függvényében
…
```

- lehetőségünk van a hívások nyomkövetésére (`Verify(…)`)

Windows Forms alkalmazások tesztelése
Példa

Feladat: Teszteljük a TicTacToe játékot.

- a korábbi teszteket kiegészítjük két új esettel:
  - betöltés (`TicTacToeGameLoadTest`), amelyben ellenőrizzük, hogy a modell állapota a betöltött tartalomnak megfelelően változott, és konzisztens maradt
  - mentés (`TicTacToeGameSaveTest`), amelyben ellenőrizzük, hogy a modell állapota nem változott a mentés hatására
- az adatelérést Moq segítségével szimuláljuk, ahol beállítjuk a betöltés visszatérési értékét, illetve ellenőrizzük, hogy valóban meghívták-e a műveleteket

Windows Forms alkalmazások tesztelése
Példa

Megvalósítás (`TicTacToeModelTest.cs`):

```
…
_mock = new Mock<IPersistence>();
_mock.Setup(mock => mock.Load(It.IsAny<String>()))
    .Returns(Enumerable.Repeat(Player.NoPlayer,9)
                       .ToArray());
    // a mock a Load műveletben minden paraméterre
    // egy üres táblának a tömbjét fogja visszaadni

_model = new TicTacToeModel(_mock.Object);
    // példányosítjuk a modellt a mock objektummal
…
```

Windows Forms alkalmazások tesztelése

Példa

Megvalósítás (`TicTacToeModelTest.cs`):

```
[TestMethod]
public void TicTacToeGameLoadTest()
{
    …
    _model.LoadGame(String.Empty);
    …
    // ellenőrizzük, hogy meghívták-e a Load
    // műveletet a megadott paraméterrel
    _mock.Verify(mock => mock.Load(String.Empty),
                 Times.Once());
}
```