



**Eötvös Loránd Tudományegyetem
Informatikai Kar**

Eseményvezérelt alkalmazások fejlesztése II

5. előadás

Windows Forms alkalmazások párhuzamosítása

Cserép Máté
mcserep@inf.elte.hu
<http://mcserep.web.elte.hu>

Készült Giachetta Roberto jegyzete alapján
<http://www.inf.elte.hu/karunkrol/digitkonyv/>

Windows Forms alkalmazások párhuzamosítása

Szinkron és aszinkron tevékenységek

- A tevékenységek végrehajtásának két megközelítése van:
 - *szinkron*: a tevékenység kezdeményezője megvárja annak lefutását
 - a hívó szál blokkolódik, amíg a tevékenység lefut
 - ha sokáig tart a tevékenység, akkor az a program felületén is észrevehető
 - *aszinkron*: a tevékenység kezdeményezője nem várja meg a lefutást, illetve az eredményt
 - a tevékenység (metódus) külön szálon fut
 - az eredményt később megkapjuk (pl. eseményen át)
 - a hívó szál nem blokkolódik, folytathatja a végrehajtást

Windows Forms alkalmazások párhuzamosítása

Példa

Feladat: Készítsünk egy grafikus felületű alkalmazást Fibonacci számok számítására.

- a Fibonacci számot egy modell állítja elő (**FibonacciGenerator**), a generáláshoz (**Generate**) a klasszikus rekurzív képletet* használjuk:

$$F(n) = \begin{cases} 1 & \text{ha } n < 3 \\ F(n-1) + F(n-2) & \text{ha } n \geq 3 \end{cases}$$

- a grafikus felületen egy listában jelenítjük meg a számokat, és egy számbeállító segítségével szabályozzuk, hányadik számra vagyunk kíváncsiak

*Valós környezetben a Fibonnaci számok a Binet formula segítségével konstans algoritmikus komplexitással előállíthatóak.

Windows Forms alkalmazások párhuzamosítása

Példa

Megvalósítás (FibonacciGenerator.cs):

```
public Int64 Generate(Int32 number) {
    if (number < 1)
        throw new ArgumentOutOfRangeException (...);
    if (number > 100)
        throw new ArgumentOutOfRangeException (...);

    if (number < 3)
        return 1;

    return Generate(number - 1)
        + Generate(number - 2);
}
```

Windows Forms alkalmazások párhuzamosítása

Aszinkron műveletek

- A grafikus felületű alkalmazások felépítésében fontos, hogy
 - gyorsan reagáljunk a felhasználói interakcióra, a felhasználói felület mindig aktív legyen
 - amennyiben egy nagyobb műveletet hajtunk végre, azt aszinkron módon, háttérben végezzük
- A háttérben futtatandó tevékenységek jelentős része (pl. fájlkezelés, hálózatkezelés) aszinkron műveletként is elérhetőek a .NET 4.5 óta:
 - ez a műveletek nevében jelzett (**Async**)
 - pl.:

```
StreamReader reader = ...;  
reader.ReadLineAsync(); // aszinkron olvasás
```

Windows Forms alkalmazások párhuzamosítása

Aszinkron műveletek

- Az szinkron műveletek eredménye bevárható egy másik aszinkron műveletben
 - aszinkron műveletet az **async** kulcsszóval hozhatunk létre
 - aszinkron műveletet bevárni az **await** utasítással tudunk

pl.:

```
private async void ReadStreamAsync(Stream str)
{
    StreamReader reader = new StreamReader(str);
    String line = await reader.ReadLineAsync();
    // aszinkron módon olvasunk, és megvárjuk
    // a művelet lefutását
    ...
}
```

Windows Forms alkalmazások párhuzamosítása

Aszinkron tevékenységek megvalósítása

- Az aszinkron műveletek alapja a *taszk* (**Task**), amely biztosítja a párhuzamos futtatást
 - a művelet tulajdonképpen taszkkal tér vissza, amely tartalmazhat eredményt is (**Task<T>**)
 - amennyiben meg szeretnénk várni a művelet eredményét, taszkot kell megadni visszatérési értéként
 - az aszinkronitást csak a megvalósításban kell jelölnünk, interfészben nem, csupán a taszk visszatérési értéket kell megadnunk
 - szinkron művelet is futtatható aszinkron módon a **Task.Run (...)** művelete segítségével, amelynek lambda-kifejezést kell megadnunk

Windows Forms alkalmazások párhuzamosítása

Aszinkron tevékenységek megvalósítása

- pl.:

```
interface IAsyncInterface {
    Task ProcessAsync();
    Task<Int32> ComputeAsync();
    // aszinkron műveletek
    // (visszatérési értékből látszik)
}

...

async Task SomeMethod(IAsyncInterface asInst) {
    Int32 result =
        await asInst.ComputeAsync();
    // eredmény bevárása
}

...
```


Windows Forms alkalmazások párhuzamosítása

Aszinkron tevékenységek megvalósítása

```
class AsyncImplementation : IAsyncInterface
{
    private void Process(); // szinkron művelet

    public async Task ProcessAsync()
    {
        await Task.Run(() => Process());
        // a tevékenység aszinkron végrehajtása
    }
    public async Task<Int32> ComputeAsync()
    {
        await Task.Run(() => { ... return value; });
    }
}
```

Windows Forms alkalmazások párhuzamosítása

Példa

Feladat: Készítsünk egy grafikus felületű alkalmazást Fibonacci számok számítására.

- a Fibonacci számot egy modell állítja elő (**FibonacciGenerator**), a generáláshoz (**Generate**) a klasszikus rekurzív képletet használjuk:

$$F(n) = \begin{cases} 1 & \text{ha } n < 3 \\ F(n-1) + F(n-2) & \text{ha } n \geq 3 \end{cases}$$

- lehetőséget adunk az aszinkron használatra is (**GenerateAsync**), lényegében egy taszkba burkoljuk a szinkron tevékenységet
- a felület így mindig aktív lesz, figyelmeztethetjük a felhasználót a tevékenységre

Windows Forms alkalmazások párhuzamosítása

Aszinkron tevékenységek megszakítása

- Az aszinkron műveleteket végrehajtása adott esetben azok teljes befejezése előtt meg kívánjuk szakítani:
 - a párhuzamos szál terminálása a háttér művelet inkonzisztens állapotban történő megszakításának kockázatával jár
 - A *taszk* alapú aszinkron eljárások támogatják az abortálási igény detektálását és kezelését:
 - `var source = new CancellationTokenSource();`
`var token = source.Token;`
`var task = new Task(() => { ... }, token);`
 - megszakítási igény jelzése a taszkon kívülről:
`source.Cancel();`
 - megszakítási igény észlelése a taszkban:
`if(token.IsCancellationRequested) { ... }`

Windows Forms alkalmazások párhuzamosítása

Példa

Feladat: Készítsünk egy grafikus felületű alkalmazást Fibonacci számok számítására.

- a Fibonacci számokat aszinkron módon egy modell állítja elő (**FibonacciGenerator**) a **Run (n)** metódussal, amely az első **n** Fibonacci számot számítja ki
- egy új Fibonacci szám előállításakor kiváltjuk a **NewResult** eseményt, az utolsó, azaz az **n.** szám előállítását követően pedig a **Ready** eseményt is
- a számítás, azaz a Fibonacci számok előállítása megszakítható a **Cancel ()** metóduson keresztül

Windows Forms alkalmazások párhuzamosítása

Példa

Megvalósítás (MainForm.cs):

```
private async void ButtonGenerate_Click(...) {  
    // aszinkron lesz az eseménykezelő  
  
    _button.Text = "Generating... Please wait.";  
    ...  
    _listBox.Items.Insert(0,  
        await _generator.GenerateAsync(...));  
    // megvárjuk a generálás eredményét  
    ...  
    _button.Text = "Generate";  
    ...  
}
```

Windows Forms alkalmazások párhuzamosítása

Példa

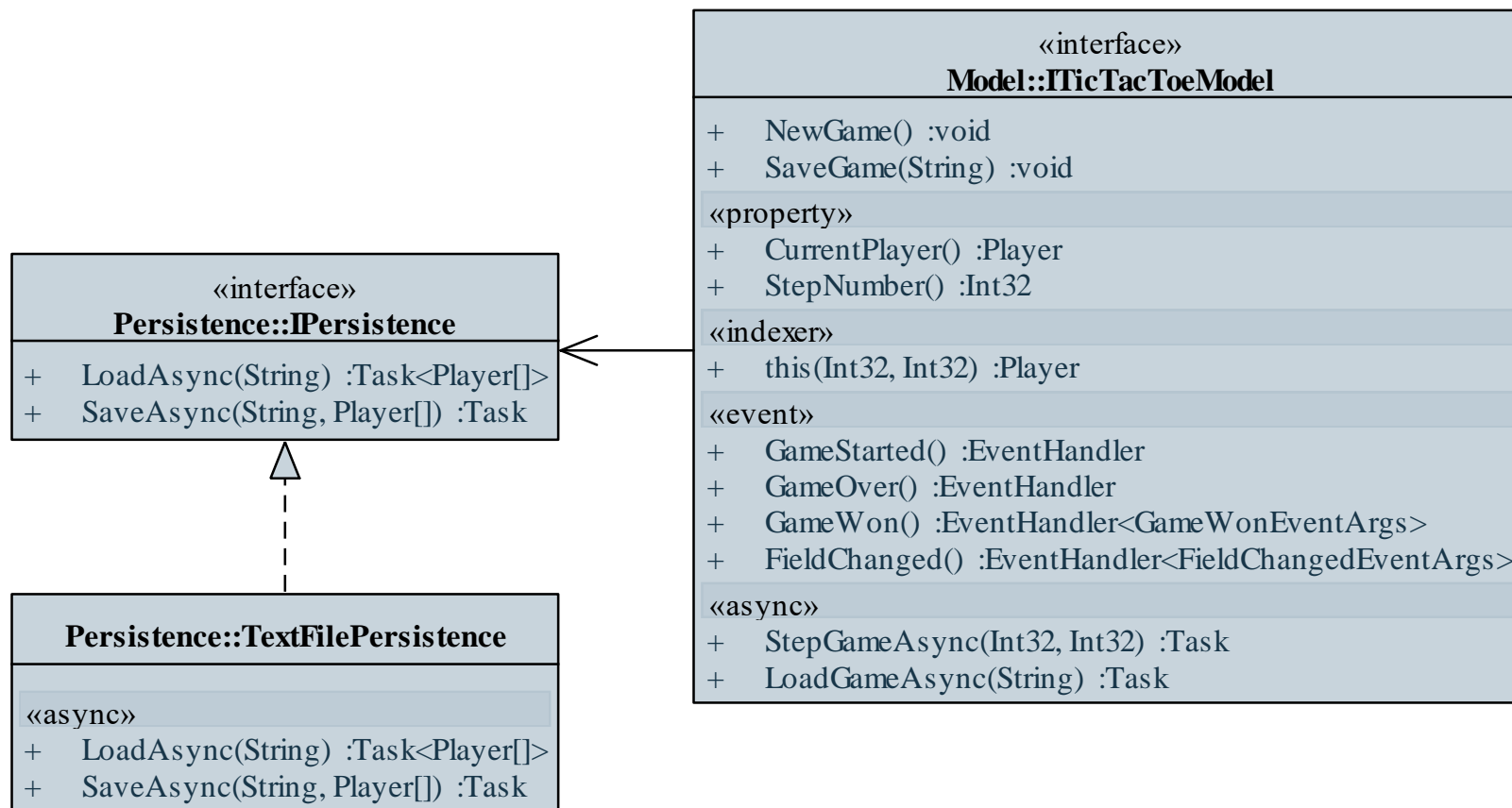
Feladat: Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- hatékonysági okokból valósítsuk meg aszinkron módon a teljes fájlkezelést, így
 - az **IPersistence** interfész **Load** és **Save** műveletei taszkkal térnek vissza
 - az **ITicTacToeModel** interfésze **LoadGame** és **SaveGame** műveletei is taszkkal térnek vissza
 - minden esetben a megvalósításban aszinkron műveleteket készítünk, és aszinkron műveleteket hívunk
 - ennek megfelelően minden felhasználáskor bevárjuk az eredményt

Windows Forms alkalmazások párhuzamosítása

Példa

Tervezés:



Windows Forms alkalmazások párhuzamosítása

Példa

Megvalósítás (TextFilePersistence.cs):

```
public async Task<Player[]> LoadAsync (String path)
...
Byte[] fileData =
    await Task.Run(() => File.ReadAllBytes(path));
    // fájl bináris tartalmának aszinkron
    // beolvasása
...
return fileData.Select(fileByte =>
    (Player)fileByte).ToArray();
}
```


Windows Forms alkalmazások párhuzamosítása

Párhuzamosítás időzítővel

- Az időzítés egy másik lehetséges formája az aszinkron tevékenység végrehajtásnak, amely a grafikus felülettől függetlenül is használható a **System.Timers.Timer** időzítővel
 - kezelhető az intervallum (**Interval**), indítás és leállítás (**Start**, **Stop**), valamint az időzített esemény kiváltása (**Elapsed**)
 - a **System.Windows.Forms.Timer** vezérlővel ellentétben párhuzamosan fut a háttérben, és nagyobb pontosságot garantál
 - hátránya, hogy amennyiben grafikus felületű alkalmazással használjuk, szinkronizálást kell végeznünk a felülettel
 - ez feloldható a vezérlő **BeginInvoke** műveletével, amely egy lambda-kifejezéssel megadott akciót (**Action**) tud futtatni a felület szálán

Windows Forms alkalmazások párhuzamosítása

Párhuzamosítás időzítővel

- Pl.:

```
Timers.Timer myTimer = new Timer(); // időzítő
myTimer.Elapsed +=
    new ElapsedEventHandler(Timer_Elapsed);
// időzített esemény
...
void Timer_Elapsed(...) {
    // itt nem használhatjuk a felületet
    BeginInvoke(new Action(() => {
        // itt már igen
        myLabel.Text = e.SignalTime.ToString();
        // kiírjuk az eltelt időt a felületre
    }));
}
```

Windows Forms alkalmazások párhuzamosítása

Példa

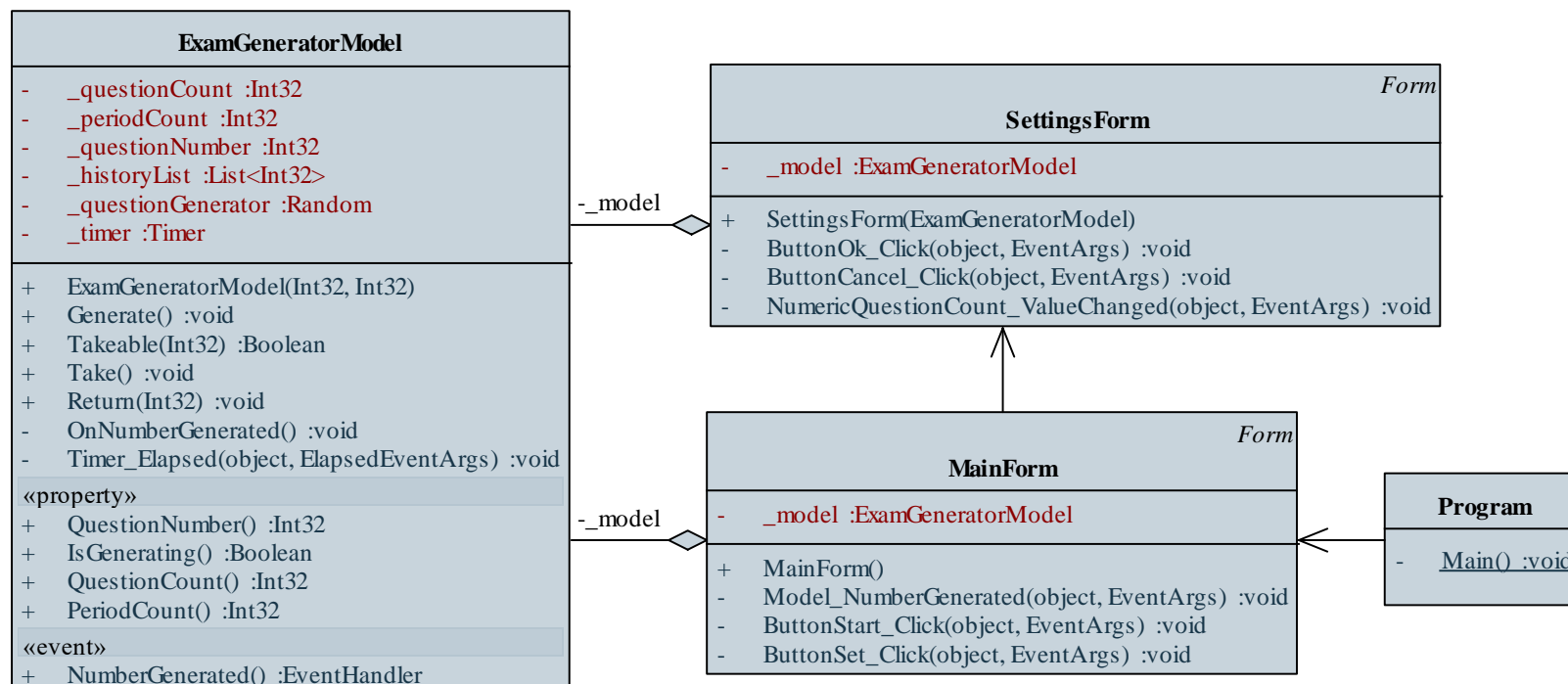
Feladat: Készítsünk egy vizsgatétel generáló alkalmazást kétrétegű architektúrában.

- a modell (**ExamGeneratorModel**) végzi a tételek generálását (**Generate**), amihez időzítőt használ, továbbá eseménnyel (**NumberGenerated**) jelzi, ha generált egy új számot
- emellett lehetőség van a tétel elfogadására (**Take**), illetve a korábban húzott tételek visszahelyezésére (**Return**)
- mindkét nézet kapcsolatban áll a modellel, a főablak az esemény hatására frissíti a megjelenítést (ügyelve a szinkronizációra)

Windows Forms alkalmazások párhuzamosítása

Példa

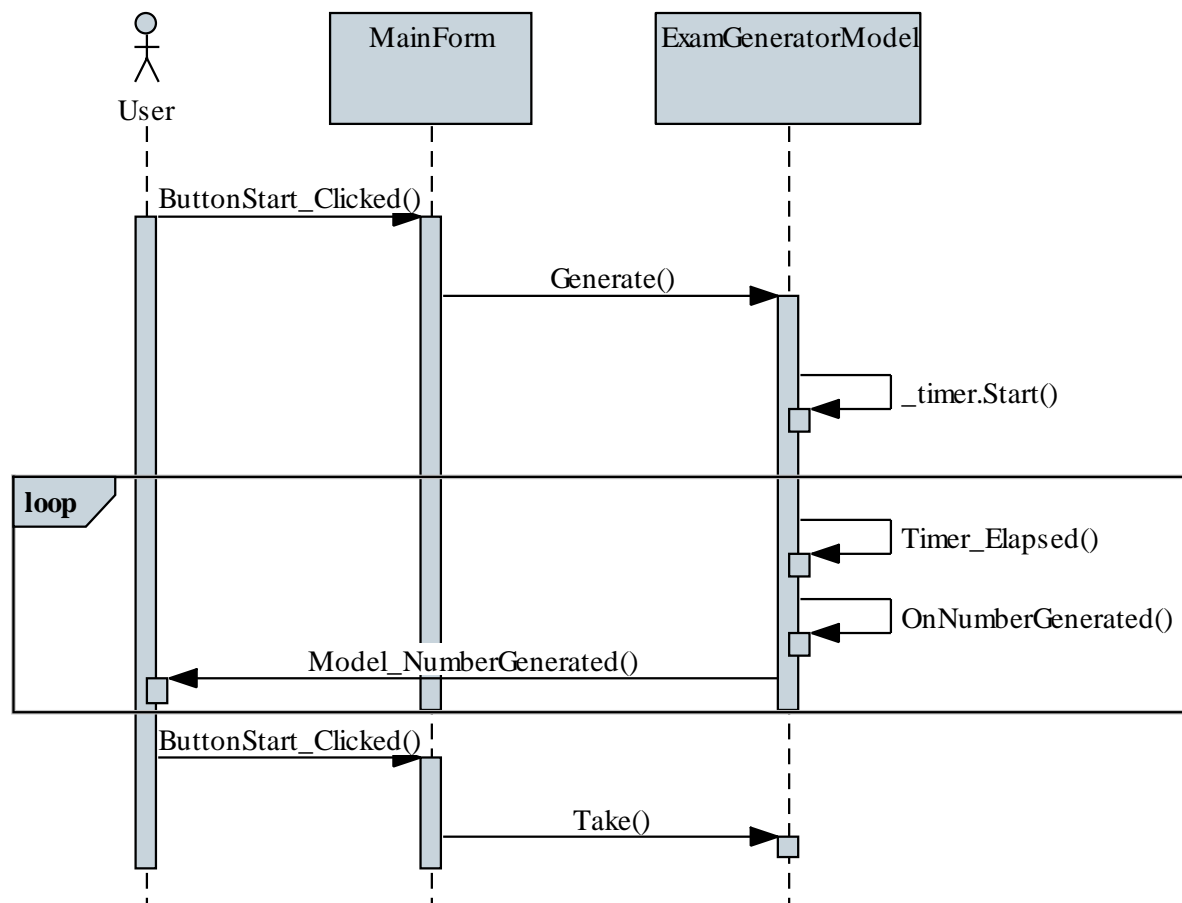
Tervezés:



Windows Forms alkalmazások párhuzamosítása

Példa

Tervezés:



Windows Forms alkalmazások párhuzamosítása

Példa

Megvalósítás (MainForm.cs):

```
public MainForm() {
    _model = new ExamGeneratorModel(10, 0);
    _model.NumberGenerated +=
        new EventHandler(Model_NumberGenerated);
    // modell eseménye
}

private void Model_NumberGenerated(object sender,
    EventArgs e) {
    BeginInvoke(new Action(() => {
        _textNumber.Text =
            _model.QuestionNumber.ToString();
    })); // szinkronizált végrehajtás
}
```