Felügyeletre tervezés

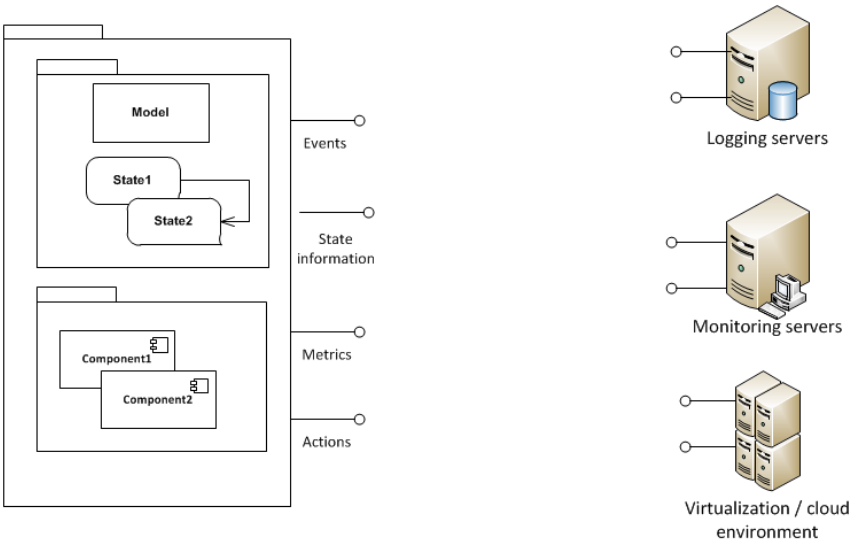
Gyakorlati útmutató
Készítette: Kocsis Imre, Micskei Zoltán
Utolsó módosítás: v1.2.3, 2016.04.15.

A segédlet célja, hogy bemutassa, hogy hogyan lehet egy alkalmazást úgy megtervezni és kibővíteni, hogy az később könnyen üzemeltethető legyen, beilleszthetővé váljon egy tipikus, jól menedzselt IT infrastruktúrába. Ezzel lehetővé válik, hogy hibák vagy terhelésváltozás esetén az alkalmazás jelzései alapján automatikus vagy félautomatikus beavatkozások történjenek, és így egy lépéssel közelebb kerülünk az autonóm IT rendszerek felé.

Az általános elvünk az, hogy először elkészítjük az alkalmazás felügyeleti modelljét. Ez általában a már meglévő struktúrát és viselkedést leíró modellek kiegészítése olyan információkkal, hogy az ott definiált adatokból mi az, ami a felügyelet szempontjából is érdekes lehet. Tipikusan a következő jellemzőket érdemes átgondolni:

- *Állapotok*: mik az alkalmazás főbb állapotai.
- *Események*: milyen eseményekről, állapotátmenetekről érdemes értesíteni a külvilágot.
- *Metrikák*: milyen konfigurációs, teljesítmény vagy szolgáltatásbiztonsági jellemzőket szeretnénk kívülről is lekérdezhetővé vagy módosíthatóvá tenni.
- *Beavatkozások*: milyen beavatkozási lehetőségeket, metódusokat akarunk az üzemeltetői rendszerből közvetlenül meghívhatóvá tenni.

Ha ezzel elkészültünk, akkor implementálni kell ezeket a jellemzőket lehetőleg valami platformszolgáltatással, majd illeszteni a meglévő IT rendszerünkhöz (lásd 1. ábra).



1. ábra: Alkalmazás csatolása az IT rendszerhez

Figyelem: A gyakorlat elvégzése előtt nézzük át a kapcsolódó előadásokat!

Tartalom

1 .NET platform

A .NET platform kiforrott technológiákat biztosít az alkalmazások felügyeleti információval való kiegészítésére:

- **WMI**: alkalmazások beállításait, állapotát, metrikáit lehet lekérdezni és megváltoztatni, beavatkozó metódusokat lehet meghívni akár távolról többféle felületen keresztül. Saját *WMI szolgáltatót* (provider) implementálhatunk menedzselt kódban is, ilyenkor jelentősen leegyszerűsödik a feladatunk a nem menedzselt verzióhoz képest.
- **Naplózás**: a platform szintű naplót az *Eseménynapló* biztosítja. Naplózó keretrendszerből is sokféle áll rendelkezésre, a segédlet az *Enterprise Library* használatát mutatja be.
- **Teljesítményszámlálók**: a teljesítményadatok tárolására és megjelenítésére szolgáló specifikus technológia a *teljesítményszámlálók* (performance counter) készítése.

A fejezet során egy egyszerű példaalkalmazás (egy számkitalálós játék) kibővítésén keresztül bemutatjuk, hogy hogyan is működnek a fenti technológiák és hogyan lehet azokat felhasználni.

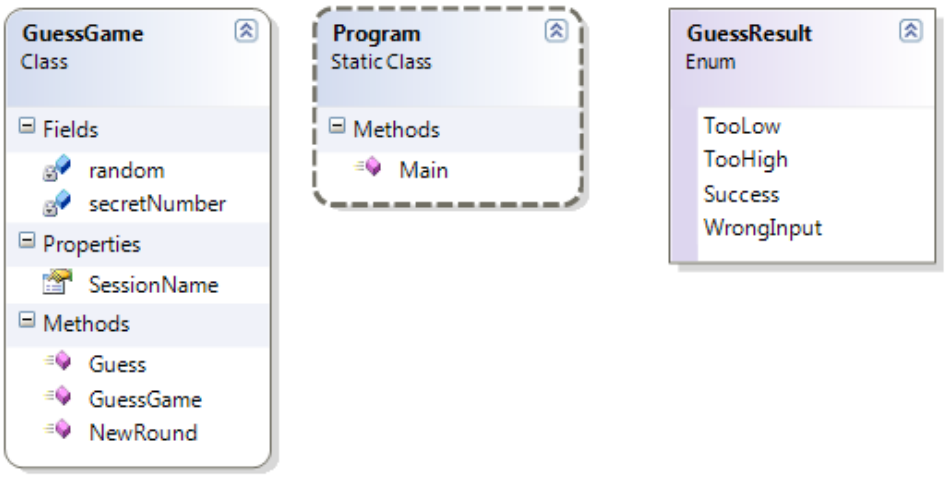
A mintakódokat 32 bites Windows 10 operációs rendszeren és 4.6-os verziójú .NET Framework segítségével próbáltuk végig.

1.1 Menedzselendő alkalmazás

A menedzselendő alkalmazás egy konzolos számkitalálós játék lesz. Egy-egy körben a program sorsol egy számot 0 és 100 között, majd a felhasználó találgathat. Minden találgatásról megmondja a program, hogy a keresett szám egyenlő, kisebb vagy nagyobb, mint az új tipp. Ha egyenlő, akkor új kör kezdődik.

1.1.1 A menedzselendő alkalmazás felépítése

A program felépítését a következő osztálydiagram szemlélteti (2. ábra).

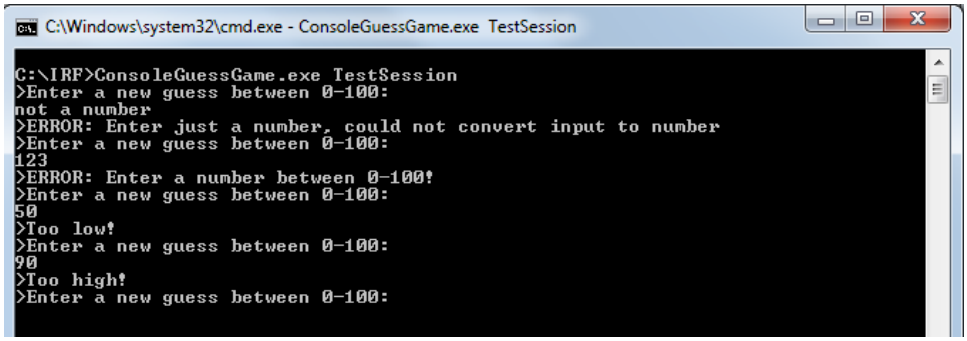


2. ábra: ConsoleGuessGame osztályai

A program elemei:

- `Program`: a belépési pontot tartalmazó osztály. A konzolról kapott paramétert (`sessionName`) ellenőrzi, majd létrehoz egy játék példányt, és a konzolról kapott bemeneteknek megfelelően vezérli azt.
- `GuessGame`: a játékot megvalósító osztály. Lehet új kört kezdeni benne, ilyenkor egy új titkos számot generál. A `Guess` metódusával pedig egy próbát lehet tenni a kitalálásra, ennek az eredménye egy `GuessResult` érték lesz.
- `GuessResult`: a próbálkozás eredményét jelzi, ez kapcsolja össze a két másik osztályt.

Az alkalmazásnak egy nagyon egyszerű konzolos felülete van, amit a következő képernyőkép szemléltet (3. ábra).



3. ábra: ConsoleGuessGame mintakimenet

Az alkalmazás egyszerűsége miatt itt a részletes felhasználói, fejlesztői és üzemeltetői ismertetőre most nem térünk ki. Az alkalmazás forráskódja megtalálható a függelékben.

1.1.2 Felügyeleti modell meghatározása

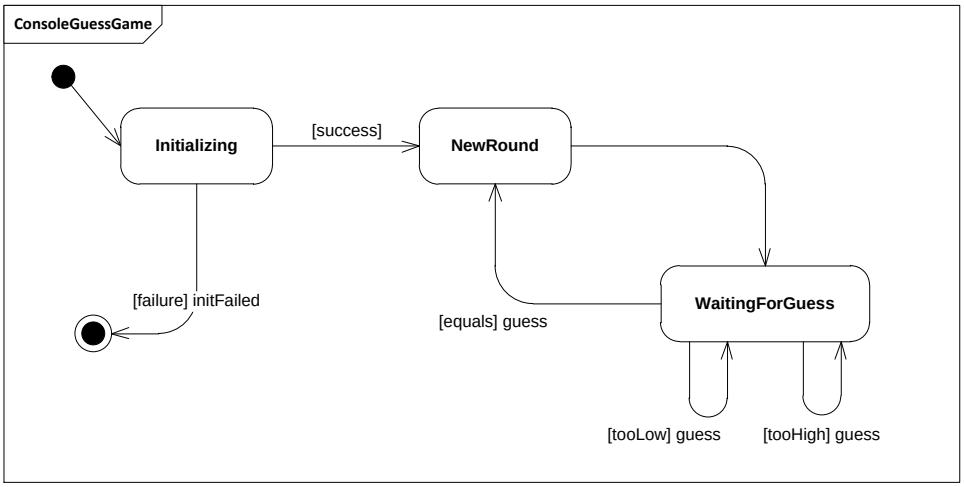
Az alkalmazáshoz először érdemes meghatározni, hogy mik a felügyeleti szempontból érdekes jellemzők.

Állapotok

A rendszer állapotgépe első közelítésben a következő módon rajzolható fel (4. ábra). Ez alapján már azonosítani tudjuk a főbb állapotokat, ezeket kéne kívülről lekérdezhetővé tenni.

Események

Az egyes fontosabb állapotátmeneteket és azok jellemzőit jó lenne naplózni is. Ehhez bővíteni kéne a fenti állapotgépet, hogy részletezzük például az egyes lehetséges hibákat. Erre itt most nem térünk ki, a kapcsolódó előadásban látható egy részletes példa. Most csak felsorolunk pár fontosabb eseményt, megadva azok azonosítóját, leírását és súlyosságát. Ezeket az eseményeket kell a fenti állapotgép finomított verziójához kötni majd, például az átmenetekhez tartozó akciók megadásával.



4. ábra: ConsoleGuessGame állapotgépe

Azonosítónak tetszőleges számot választhatunk, ám érdemes mondjuk valami két- vagy háromjegyű számnál indulni, hogy később az implementáció során ne ütközzön a naplózó keretrendszer valami alapértelmezetten választott azonosítójával. Lehet valamilyen strukturáltabb azonosítókat is használni, például itt most azt az elvet követjük, hogy a 100-zal kezdődő azonosítók sikeres, míg a 200-zal kezdődők sikertelen végrehajtást jelölnek.

1. táblázat: ConsoleGuessGame néhány eseménye

ID	Súlyosság	Szöveg / Definíció
101	Information	„ConsoleGuessGame initialized successfully.” Az alkalmazás sikeresen befejezte a bemeneti paraméterek feldolgozását, létrehozta a szükséges objektumokat, és kész egy új játékot indítani.
201	Critical	„Invalid startup parameters: {0}” Nem megfelelő számú vagy értékű bemeneti paramétert kapott az alkalmazás. (Ezt az eseményt később még érdemes tovább finomítani.)
102	Verbose	„New guess received: {0}.” Új próbálkozás érkezett, az esemény szövegének része a kapott érték.

Metrikák

Miután megfogalmaztuk a rendszer működését, határozzuk meg, hogy mik azok a jellemzők, amiket érdemes számon tartani és elérhetővé tenni az alkalmazásban. Az alkalmazás egyszerűsége miatt itt most két triviálisan számolható metrika szerepel, bonyolultabb példák szerepelnek az előadás anyagában.

Arra figyeljünk metrikák esetén, hogy intenzíven használt alkalmazás esetén egy-egy metrika értéke könnyen túlcsordulhat vagy feleslegesen sok erőforrást foglal, így sok esetben érdemes valami számított jellemzőt (átlag, minimum, utolsó tíz érték, stb.) eltárolni az összes adat helyett.

2. táblázat: ConsoleGuessGame két metrikája

Metrika	Leírás	Számítási módszer
TotalGuesses	Az aktuális körben az eddigi összes találgatás száma. Tartalmazza a nem megfelelő és jó tartományba eső találgatásokat is.	Egy egész számláló, melyet eggyel növelünk minden egyes találgatáskor. Új kör kezdése esetén nullázzuk. Nem tartalmazza a nem egész számként megadott találgatásokat.
TotalFaultyGuesses	Az aktuális körben az összes nem megfelelő (nem az elvárt tartományba) eső találgatások száma.	Egy egész számláló, melyet eggyel növelünk minden egyes olyan találgatáskor, ami nem az elvárt számtartományban van. Új kör kezdése esetén nullázzuk. Nem tartalmazza a nem egész számként megadott találgatásokat.

Akciók

Végezetül határozzuk meg, hogy milyen beavatkozási lehetőségek vannak, amiket az alkalmazás működése során az üzemeltetői rendszerből kell tudnunk könnyen kiadni.

3. táblázat: ConsoleGuessGame beavatkozási lehetőségei

Akció	Definíció
NewRound	Új számot sorsol, és nullázza az aktuális körre vonatkozó metrikákat.

1.2 WMI szolgáltató készítése

Ha a saját alkalmazásunkat ki szeretnénk egészíteni, hogy konfigurációs és monitorozási információkat szolgáltasson Windows platformon, akkor erre az ajánlott módszer egy *WMI szolgáltató*[1] (provider) készítése. A .NET Frameworkben már egészen jó támogatás van arra, hogy mindezt menedzselt kódban írjuk meg, az úgynevezett *WMI Provider Extensions* [1] segít ebben (a `System.Management.Instrumentation` névtér elemei). Az MSDN dokumentációban megtaláljuk a használandó osztályok részletes leírását és példakódokat ezek használatára. Egy részletesen elmagyarázott példakód található itt [3]. A következőkben csak a legfontosabb elemeket tekintjük át.

Megjegyzés: A Windows 8-ban megjelent új Management Infrastructure (MI) technológiához .NET-ben csak a kliens API érhető el, amivel CIM lekérdezéseket lehet megvalósítani (a `Microsoft.Management.Infrastructure` névtérben), a szolgáltató készítéséhez csak C++ API-t ajánlanak ki.

1.2.1 Tervezői döntések WMI szolgáltató készítése esetén

Saját WMI szolgáltató készítése esetén az első kérdés, amit el kell döntenünk, hogy milyen úgynevezett “hosting modelt” használunk:

[1] A szakkifejezések magyar megfelelői a http://www.microsoft.com/Language adatbázisból vannak.

- *Folyamatbeli (In-process)*: A szolgáltatót egy DLL-ben implementáljuk, amit a *Global Assembly Cache*-be (GAC) kell rakni. A szolgáltató a WMI kiszolgáló folyamatában fut, az folyamatosan elérhető.
- *Leválasztott (Decoupled)*: A szolgáltató kódja az alkalmazás kódjába kerül, így az csak akkor érhető el, ha az alkalmazásunk fut. Azonban mivel az alkalmazással együtt fut, annak állapotát és pillanatnyi értékeit is könnyedén ki tudja olvasni.

A következő választási lehetőségünk, hogy hány példány lehessen a szolgáltatóból:

- *Singleton*: a szolgáltatóból csak egy példány lehet, a WMI keretrendszer gondoskodik a példány életciklusának kezeléséről.
- *Multi-instance*: a WMI szolgáltatóból több példány lehet, ezeket kulcsokkal lehet megkülönböztetni (hasonlóan, mint ahogy azt a beépített WMI osztályok esetén láttuk, pl. a `Win32_Processor` lekérdezése során).

Megjegyzés: bár csábító a Singleton szolgáltató használata, mert könnyebbnek tűnik implementálni, de az elmúlt évek tapasztalata azt mutatja, hogy elég sok gond van vele (pl. máshogy kell regisztrálni, nehéz rajta metódust hívni, stb.).

Leválasztott, többpéldányos szolgáltató esetén még el kell dönteni, hogy a WMI rendszert hogyan tudatjuk az egyes példányok létezéséről. A két lehetséges módszerről sajnos elég kevés információ van, pedig lényeges következményei vannak annak, hogy melyiket választjuk. A hivatalos dokumentáció[2] ennyit mond:

- *Publish/revoke*: „In the publish/revoke model, the WMI infrastructure provides default behavior for many of the methods you have to write yourself in the callback method. These include the enumeration and bind methods. In this model, the application creates instances and publishes them. The application is responsible for ensuring that the key properties of the classes are respected. The application is also responsible for deleting instances."
- *Callback*: „In the callback model, the WMI infrastructure expects the application to have methods that handle enumeration, binding and any other methods required to implement the functionality of the provider. It calls into the application for this functionality and fails if it does not exist or is not implemented properly. The application registers the type of its WMI classes with the infrastructure by calling RegisterType and indicates that it no longer wants the WMI classes exposed by calling UnregisterType."

Mindkét módszer használata esetén a tényleges funkcionalitást (WMI-ból elérhető tulajdonságok és metódusok) ugyanúgy kell implementálni, például ugyanaz a MOF fájl generálódik belőle.

A különbség az, hogy *Publish* esetén a szolgáltató .NET-es osztályból mindig az alkalmazás saját kódja példányosít és ezeket az objektumokat tesszük közzé a WMI keretrendszerben. A WMI keretrendszer utána mindig ezeken az objektumokon hívja meg a kért tulajdonságokat vagy metódusokat.

[2] A `System.Management.Instrumentation.InstrumentationManager` osztály leírásából.

A *Callback* módszer használata esetén viszont egy statikus *Enumerator* metódust kell készítenünk a szolgáltató kódjában, és azt kell megjelölni a `ManagementEnumerator` attribútummal. Ennek kell visszaadnia a szolgáltató összes példányához tartozó objektumokat. A WMI keretrendszertől érkező hívások először ezt a metódust fogják meghívni. Ezen kívül meg kell adni egy úgynevezett *Bind* metódust is, ami egy konkrét kulcsú példányt visszaad. Ez lehet egy konstruktor vagy egy statikus metódus is, csak arra kell odafigyelni, hogy a paraméterei ugyanolyan nevűek és típusúak legyenek, mint a szolgáltató kulcsai. A metódust a `ManagementBind` attribútummal kell megjelölni. Fontos, hogy a metódusnak be kell állítania a példány összes tulajdonságát a megfelelő értékűre is. A WMI keretrendszer a működése során hívhatja ezt a *Bind* metódust is, így ha ez egy konstruktor, akkor menet közben új .NET objektumokat hozhat létre azokon kívül, amit mi az *Enumerator* segítségével visszaadtunk.

A különbséget jól szemlélteti a következő naplórészlet (4. táblázat és 5. táblázat).

4. táblázat: Publish módszer szemléltetése

Metódushívás	Objektum ID	Szál ID
Szolgáltató konstruktora	O1	3276
Publish hívás (Main-ben)		3276
Kulcs tulajdonság lekérdezése	O1	2432
Beavatkozó metódus	O1	2432

Mindkét esetben egy példányt hozunk létre a szolgáltatónkból, majd meghívunk rajta egy beavatkozó metódust. *Publish* esetén a saját alkalmazásunkban létrehozzuk a szolgáltató példányát, majd közzétesszük. Amikor kívülről WMI-on keresztül meghívjuk a WMI-on elérhetővé tett metódust, akkor ezen az objektumon kérdeződik le a kulcs, majd hívódik meg a metódus. Figyeljük azonban meg, hogy a WMI kérés kiszolgálása már egy másik szálon megy!

5. táblázat: Callback módszer szemléltetése

Metódushívás	Objektum ID	Szál ID
RegisterType hívás (Main-ben)		2964
Enumerator hívás (statikus metódus)		1012
Kulcs tulajdonság lekérdezése	O2	1012
Bind metódus hívás (szolgáltató konstruktora)	O3	1012
Beavatkozó metódus	O3	1012

Callback esetén a következő történik. A hívás első lépéseként meghívódik az *Enumerator* metódus, amiben visszaadjuk a szolgáltató példányait. Ezeknek megvizsgálja a kulcs tulajdonságait. Utána a *Bind* metódust meghívja azzal a kulcs értékkel, amin a beavatkozó

metódust akarjuk végrehajtani. Ezzel viszont már egy új objektumot hozott létre, mivel a konkrét példában a *Bind* metódusként konstruktort jelöltünk meg. A beavatkozó metódust ezen az új objektumon hívja már meg. Ennek tehát az a következménye, hogy az adatokat nem szabad a szolgáltató osztályban objektum szinten tárolni, hisz a lekérdezés során több különböző példány is létrejöhet.

Nézzük akkor most meg egy példán keresztül, hogy hogyan kell elkészíteni egy leválasztott, többpéldányos WMI szolgáltatót.

1.2.2 A WMI szolgáltató kódjának megírása

Ha végiggondoltuk az alkalmazásunk felügyeleti modelljét, azonosítottuk a később lekérdezendő tulajdonságokat, akkor nekiállhatunk a kód kiegészítésének, hogy WMI szolgáltatóként funkcionálhasson.

A *.NET WMI Provider Extensions* használata nagyobbrészt abból áll, hogy megfelelő attribútumokkal felcímkézzük az alkalmazásunkat. A menedzsment információk tárolását és lekérdezésének módját azonban érdemes előtte végiggondolni.

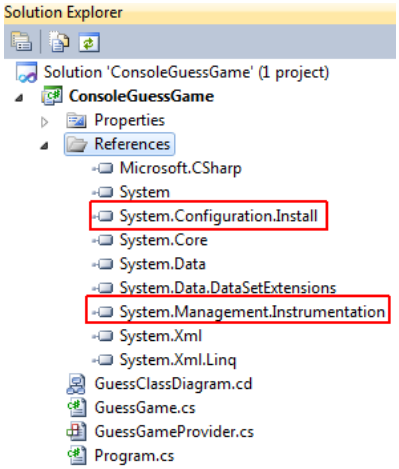
Itt is sokféle tervezési mintát és módszert követhetünk, felcímkézhetjük a meglévő osztályokat, kigyűjthetjük a WMI szolgáltatóval kapcsolatos részeket külön helyre stb. Mindegyik módszernek megvannak az előnyei és hátrányai. Itt most a példában azt az elvet követjük, hogy egy külön osztályt hozunk létre, ami a menedzsment információkat kezeli.

A következő tervezői döntés, hogy hogyan kapcsoljuk ezt az osztályt a meglévő kódhoz. Remélhetőleg erre is mindenkinek többféle ötlete lenne:

- Az üzleti objektumok értesíthetik a menedzsment objektumot esemény esetén, és az tárolja el az információkat (*push*),
    - ilyenkor fontos döntés, hogy hogyan kapnak referenciát a menedzsment objektumra: konstruktor paraméterként megkapják, valami statikus osztálytól ezt lekérdezhetik, egy *Factory* segítségével legyártatják, stb.
- Külső kérés esetén a menedzsment objektum kérdez körbe, és gyűjti be a neki szükséges információkat (*pull*),
    - ebben az esetben is meg kell határoznunk, hogy hogyan éri el a menedzsment objektum azt, akihez majd fordulnia kell.

A fenti döntést nyilván befolyásolja az is, hogy a *Publish* vagy *Callback* módszert használunk, mert *Callback* esetén nem célszerű, ha a menedzsment objektum tárol adatokat.

A példában a *Publish* módszert alkalmazzuk. Továbbá az üzleti objektumok tárolják az információt, és a menedzsment objektum igény esetén ezeket kérdezi le. Ez most a jelen kis alkalmazásunkban, ahol egy üzleti osztály van, jó megoldás, komplexebb alkalmazás esetén, nem biztos, hogy ez a legcélszerűbb architektúra.



5. ábra: Szükséges referenciák hozzáadása

1. Adjuk hozzá a projektünkhöz a `System.Management.Instrumentation` és a `System.Configuration.Install` szerelvényt referenciaként (5. ábra).
2. Hozzunk létre egy `GuessGameProvider` osztályt, és adjuk hozzá a következő névtér használatot:

```
using System.Management.Instrumentation;
```

3. Jelöljük meg, hogy az alkalmazás szerelvénye egy WMI szolgáltatót implementál, erre a `WmiConfiguration` attribútum való. Ezt az osztályon kívül kell valahol elhelyeznünk, célszerű a *using* részek után.

```
[assembly: WmiConfiguration("root/irf",
 HostingModel = ManagementHostingModel.Decoupled, IdentifyLevel = false)]
```

   - Az első paraméter azt adja meg, hogy milyen névtérbe kerüljön majd a WMI szolgáltató.
   - *HostingModel*: itt lehet in-process vagy decoupled mód közül választani.
   - *IdentifyLevel*: ez a megszemélyesítést (impersonation) engedélyezi vagy tiltja, tehát, hogy a WMI kérést indító felhasználó jogaival hajtsa-e végre a WMI szolgáltató a műveleteket vagy nem[3].
   - Vannak még további tulajdonságai is, ezek az MSDN leírásban megtalálhatóak.
4. Ahhoz, hogy beregisztráljuk a szolgáltatónkat a WMI keretrendszerbe, szükségünk van a Managed Object Format (MOF) leírására. Ezt szerencsére nem kell kézzel elkészíteni, ha létrehozunk egy, a `DefaultManagementInstaller` osztályból származó saját osztályt, akkor ez később az *installutil.exe* segítségével elvégez minden szükséges teendőt.

```
[System.ComponentModel.RunInstaller(true)]
public class TheInstaller : DefaultManagementInstaller { }
```

[3] Az IdentifyLevel nem igazán úgy működik, ahogy a dokumentációban le van írva, lásd [2].

5. Meg kell jelölnünk, hogy melyik osztály valósítja meg a szolgáltatót. Erre a `ManagementEntity` attribútum való.

```
[ManagementEntity(Name = "IRF_GuessGame")]
[ManagementQualifier("Description", Value = "Provider for GuessGame.")]
class GuessGameProvider
```

   - *Name*: a szolgáltató osztály neve, ahogy majd megjelenik a WMI-ban.
   - *Singleton*: itt lehetne beállítani, hogy singleton legyen a szolgáltató, ezt most nem tesszük meg.

   Figyeljük meg, hogy az osztályhoz hozzáraktunk még egy `ManagementQualifier` attribútumot, ez egy *CIM minősítőre* (qualifier) képződik majd le. Ennek az értéke most egy leírás, de tetszőleges más minősítő is megadható (pl. mértékegység, verzió stb.). Célszerű legalább a leírást megadni majd minden későbbi elemhez (tulajdonság, metódus stb.) is.

6. Mivel ez egy többpéldányos WMI szolgáltató lesz, meg kell adni, hogy mik lesznek a kulcs tulajdonságai a szolgáltatónknak. Lehetne összetett kulcsot is használni, itt most nekünk egy elindított játékpéldányt egyértelműen azonosít a parancssori paraméterként megadott `sessionName` értéke, úgyhogy ezt fogjuk használni. Azt a tulajdonságot, amivel a kulcs lekérdezhető, `ManagementKey` attribútummal kell megjelölni.

```
private string sessionName;

[ManagementKey]
public string SessionName
{
        get { return this.sessionName; }
}
```

7. Mivel a *Publish* módszert használjuk, nincs szükség se a `ManagementBind`, se a `ManagementEnumerator` attribútumra.

8. Az alkalmazásunkban még létre kell hozni valahol a WMI szolgáltatót, majd beregisztrálni. Rakjuk ezt be a `Program` osztályunkba.

```
GuessGameProvider provider = new GuessGameProvider(args[0]);
InstrumentationManager.Publish(provider);
```

   Figyelem: ez az a rész, amit éles alkalmazás esetén jól át kell gondolni. Egyrészt el kell dönteni, hogy hány darab példányra van szükségünk a szolgáltatóból. Másrészt valahogy garantálni kell, hogy a kulcs egyedi legyen. Jelen példa erre nem figyel, ha másik példányt indítunk a programunkból, az beregisztrálhat egy ugyanilyen nevű példányt.

9. Ezzel a minimális váza elkészült a szolgáltatónak. Most már csak az alkalmazás-specifikus metrikákat kéne lekérdezhetővé tenni. Erre a következő attribútumok szolgálnak:

   - `ManagementProbe`: egy csak olvasható tulajdonságot jelöl,

- `ManagementConfiguration`: írható és olvasható tulajdonságot jelöl,
- `ManagementTask`: WMI-ból hívható metódust jelöl.

10. Készítsünk a `Program` osztályban egy statikus tulajdonságot, amin keresztül a `GuessGame` aktuális példánya elérhető. Ezen keresztül fogja akkor a szolgáltató elérni az alkalmazás aktuális állapotát.

    Figyelem: komplexebb alkalmazás esetén más módszer lehet a célszerűbb az üzleti logika és a szolgáltató összekötésére.

11. Adjunk hozzá egy csak olvasható tulajdonságot, amit WMI-on keresztül el lehet érni.

```
[ManagementProbe]
public int TotalGuesses
{
    get
    {
        if (Program.Game != null)
        {
            return Program.Game.TotalGuesses;
        }
        else
        {
            return 0;
        }
    }
}
```

12. Hozzunk létre egy WMI-on keresztül hívható metódust is.

```
[ManagementTask]
public void NewRound()
{
    if (Program.Game == null)
    {
        //TODO raise or log error
    }
    else
    {
        Program.Game.NewRound();
    }
}
```

13. Most már szolgáltat némi információt a WMI szolgáltatónk, akár használatba is vehetjük.
14. **Figyelem**: egy fontos dologról csúnyán elfelejtkeztünk. Onnantól kezdve, hogy az alkalmazáson belül fut a WMI szolgáltató, automatikusan többszálúvá vált a programunk, ugyanis a WMI kérések kiszolgálása másik szálon zajlik. Tehát a kölcsönös kizárás megvalósítására oda kell figyelnünk, például megfelelő zárolás használatával.

1.2.3 A WMI szolgáltató telepítése

Miután elkészült a szolgáltató kódja, telepíteni kell még azt.

1. Mivel létrehoztunk benne egy telepítőt, ezért ezt az `installutil.exe` megoldja. Futtassuk is le (itt most a kimenetből csak a legfontosabb részeket tartottuk meg)!

   Figyelem: ezt rendszergazda jogú parancssorban kell kiadni!

```
C:\IRF>c:\Windows\Microsoft.NET\Framework\v4.0.30319\InstallUtil.exe ConsoleGuessGame.exe

Running a transacted installation.

Beginning the Install phase of the installation.
See the contents of the log file for the C:\IRF\ConsoleGuessGame.exe assembly's progress.
The file is located at C:\IRF\ConsoleGuessGame.InstallLog.
Installing assembly 'C:\IRF\ConsoleGuessGame.exe'.
**** WMI schema install start ****
**** WMI schema install start ****
**** WMI schema install end ****

The Install phase completed successfully, and the Commit phase is beginning.
Microsoft (R) MOF Compiler Version 6.1.7600.16385
Parsing MOF file: C:\Windows\system32\wbem\ConsoleGuessGame_v4.0.30319.mof
MOF file has been successfully parsed
Storing data in the repository...
Done!
```

   Tehát létrehoz egy MOF fájlt, majd beregisztrálja azt a WMI-ba. Az `installutil` részletes naplófájlokat készít, hiba esetén ezeket érdemes megnézni.

2. Nézzük meg a létrehozott MOF fájlt, ezzel tudunk meggyőződni, hogy minden lényeges elem átkerült a szolgáltatónkból. Itt most megint csak a leglényegesebb elemeket tartottuk meg.

```
#pragma namespace("\\\\.\\root\\irf")

class WMI_extension : __Win32Provider{
 string Name = NULL;
 string CLSID = "{2A7B042D-578A-4366-9A3D-154C0498458E}";
 uint32 Version = 1;
 string HostingModel = "Decoupled:COM";
 string SecurityDescriptor = NULL;
 string AssemblyPath;
 string AssemblyName;
 string CLRVersion;
};

instance of WMI_extension{
 AssemblyName = "ConsoleGuessGame, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null";
 AssemblyPath = "file:///C:/IRF/ConsoleGuessGame.exe";
 CLRVersion = "v4.0.30319";
 CLSID = "{2A7B042D-578A-4366-9A3D-154C0498458E}";
 HostingModel = "Decoupled:COM:FoldIdentity(FALSE)";
 Name = "ConsoleGuessGame, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null";
};

[dynamic(TRUE) : ToInstance , provider("ConsoleGuessGame, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=null") , Description("Provider for the GuessGame
application.") : ToSubClass ToInstance]
class IRF_GuessGame{
 [read(TRUE) , key(TRUE)] string SessionName;
 [read(TRUE)] sint32 TotalGuesses;
```

```
 [read(TRUE)] sint32 TotalFaultyGuesses;
 [implemented(TRUE)] void NewRound();
};
```

Készült tehát egy `WMI_extension` példány, ami a szolgáltatóval kapcsolatos általános információkat tárolja, valamint megjelent az `IRF_GuessGame` osztályunk is a szükséges tulajdonságokkal.

3. Ha valamiért el szeretnénk később távolítani a WMI szolgáltatót a rendszerből, akkor azt is `installutil.exe` segítségével tehetjük meg a `/u` kapcsoló megadásával.
4. Ha módosítunk valamit az alkalmazáson, ami a definiált WMI tulajdonságokat is érinti, akkor az `installutil.exe` programot újra le kell futtatnunk, hisz a rendszerbe beregisztrált MOF leíró nem frissül automatikusan a program újrafordításakor.

Ezzel végeztünk is a telepítéssel, a WMI szolgáltató mostantól használható.

1.2.4 A WMI szolgáltató használata

Nézzük meg, hogy hogyan kérdezhető le a frissen elkészült szolgáltatónk. Ehhez PowerShell lekérdezéseket fogunk használni.

1. Kérdezzük le az osztályunk példányait!

```
PS C:\> Get-CimInstance IRF_GuessGame -Namespace root\irf
PS C:\>
```

Nem kaptunk semmit vissza, ugyanis leválasztott szolgáltatót készítettünk, így ha nem fut az alkalmazásunk, akkor nem is lesz WMI osztály példány hozzá.

2. Próbáljuk elindítani az elkészült alkalmazást:

```
C:\IRF>ConsoleGuessGame.exe TestSession

Unhandled Exception: System.Management.Instrumentation.WmiProviderInstallationException:
Exception of type 'System.Management.Instrumentation.WMIInfraException' was thrown.
   at System.Management.Instrumentation.InstrumentationManager.Publish(Object value)
   at IRF.ConsoleGuessGame.Program.Main(String[] args)
```

Mi a tanulság ebből?

- A Publish hívást érdemes lett volna try-catch blokkba rakni, és megakadályozni, hogy nem kezelt kivétel jusson ki a legfelső szintre.
- A Publish-t nem hívhatja meg akárki, rendszergazdai jogok kellenek ahhoz, hogy a WMI tárházba (repository) valaki elemeket rakjon be.

Vagy az alkalmazásunkat rendszergazdaként futtatjuk ezután (ami – lévén ez egy egyszerű kliensalkalmazás – nem elfogadható megoldás), vagy megmondjuk, hogy más is publikálhat ilyen szolgáltatót. Ehhez a szolgáltatóhoz tartozó `WMI_extension` osztály `SecurityDescriptor` tulajdonságát kell módosítani. Ez egy szöveges formátumban várja a biztonsági leírót, ennek a leírását lásd [4]. Jelen esetben használhatjuk a következőt:

```
O:%G:%D:(A;;0x1;;;%)
```

ahol a % helyére kell azt a SID-et írni, akinek engedélyezni akarjuk a publikálást. Ez lehet felhasználó vagy csoport SID-je is. Mi most a példában használjuk a beépített Everyone csoporthoz tartozó speciális karaktersort (WD). A következő PowerShell utasítás beállítja a SecurityDescriptor megfelelő értékét (rendszergazdaként kell végrehajtani!).

```
Get-Wmiobject -class WMI_Extension -namespace root\irf |
 % { $_.SecurityDescriptor = "O:WDG:WDD:(A;;0x1;;;WD)"; $_.Put() }
```

Ezt a beállítást minden telepítés után újra el kell végeznünk, hisz a telepítés újra létrehozza a `WMI_Extension` osztály definícióját a WMI-ban.

3. Most már el tudunk indítani két példányt az alkalmazásból, és le is tudjuk kérdezni a WMI példányokat.

```
PS C:\> Get-WmiObject IRF_GuessGame -Namespace root\irf

__CLASS            : IRF_GuessGame
__RELPATH          : IRF_GuessGame.SessionName="GameSession"
__NAMESPACE        : root\irf
__PATH             : \\WIN-VM\root\irf:IRF_GuessGame.SessionName="GameSession"
SessionName        : GameSession
TotalFaultyGuesses : 1
TotalGuesses       : 3

__CLASS            : IRF_GuessGame
__RELPATH          : IRF_GuessGame.SessionName="TestSession"
__NAMESPACE        : root\irf
__PATH             : \\WIN-VM\root\irf:IRF_GuessGame.SessionName="TestSession"
SessionName        : TestSession
TotalFaultyGuesses : 0
TotalGuesses       : 1
```

A `__PATH` tulajdonságban látszik a példány elérési útja: a számítógép, WMI névtér, osztály és kulcsok összessége.

Ezen kívül látszik, hogy a WMI-ból elérhető tulajdonságokat is megjeleníti szépen.

4. Most már csak egy metódus meghívása van hátra. Használhatjuk erre az `Invoke-CimMethod` cmdletet, de ha nem kell bonyolult paramétereket átadni, akkor megfelel a következő módszer is.

```
(Get-WmiObject IRF_GuessGame -Namespace root\irf -filter "sessionname =
'TestSession'").NewRound()
```

Sajnos a `Get-CimInstance` új cmdlettel nem látszik legtöbbször a WMI metódus.

1.2.5 Tipikus hibák

Sajnos a WMI szolgáltatók által visszaadott hibakódokról kevés dokumentáció van.

- Miután telepítettük az `installutil.exe` segítségével a saját szolgáltatónkat, érdemes megnézni a MOF állományában, hogy tényleg átkerült-e az összes tulajdonság és metódus, amit a kódban definiáltunk.
- Az MSDN dokumentációban találhatunk néhány jó tanácsot [5].
- Be lehet kapcsolni a *Tracing* szintű naplózást a WMI-ra, ez ilyenkor minden műveletet részletesen naplóz az Eseménynaplóba [6]. Ez az összes műveletet kezdetét és végét naplózza, ami elég sok adat lehet, így azért néha nem könnyű kiigazodni benne.
- Ha a `WmiConfigurationAttribute`-ot hiányolja a fordító, akkor a `System.Core` szerelvényre kell referenciát hozzáadni (a `WmiConfigurationAttribute` MSDN-es leírásában az Assembly résznél szerepel az, hogy melyik szerelvény tartalmazza).
- *"Provider is not capable of the attempted operation"* hiba (kód: 0×80041024): nincs megadva `ManagementBind` attribútum egyik metódusra sem, vagy az nem publikus, vagy nem konstruktor vagy statikus metódus, így a WMI nem tud mit meghívni, amikor el szeretné érni a szolgáltatónkat. Ha `InstumentationManager.RegisterType` típusú regisztrálást használunk, akkor pedig léteznie kell egy `ManagementEnumerator` attribútummal dekorált statikus metódusnak is.
- A `WMIInfraException` jogosultság gond esetén szokott például előjönni, ellenőrizzük, hogy rendszergazdaként futtatva is ez a hiba-e.
- A `WmiProviderInstallationException` azt jelezheti, hogy nem regisztráltuk be a WMI-ba a szolgáltatónkat az `installutil.exe` segítségével.
- Ha nem adunk meg kulcsot a szolgáltatónkban `ManagementKey` segítségével, attól még települ rendesen, csak majd lekérdezésekor nem kapunk vissza egy példányt sem.
- Egy viszonylag aljas hiba, hogy előfordulhat olyan, hogy a WMI számára publikált metódusunk osztály szintjén még látszik, azonban példány szinten már nem. Ilyenkor a MOF fájlban ott van a metódus definíciója, azonban a konkrét WMI objektumon már nem tudjuk meghívni PowerShellből. Ilyenkor zárjuk be a PowerShell ablakot, és indítsunk egy újat, abból már látszani fog példány szinten is a metódus[4].
- Ha `Publish()` segítségével tesszük elérhetővé egy leválasztott szolgáltató példányait, és több különböző folyamatban fut a programunk, akkor találkozhatunk a következő jelenséggel. Ha meghívunk az egyiken egy metódust, akkor egy üres leírású `WmiMethodException` lesz az eredmény, a metódushívás nem kerül végrehajtásra és valamelyik másik folyamat a programunkból terminálódik (más is találkozott már ezzel, lásd [7]). Különösen széppé teszi a hibát, hogy ha a program másik példányát Visual Studio debuggerben futtatjuk, akkor a hiba nem jelentkezik.
- Ha a `Get-WmiObject` lekérdezés során `GetWMICOMException` hibát kapunk, és a menedzselt alkalmazás is leáll, akkor tipikusan valamelyik `ManagementProbe` lekérdezése során dobódik egy nem kezelt kivétel. Pontosan információt úgy lehet kapni, ha a menedzselt alkalmazásunkat debuggerben indítjuk, és akkor ott meg lehet találni, hogy melyik utasítás végrehajtása során keletkezett kivétel.

[4] Köszönet Szabó Bálintnak a probléma megoldásáért.

- Figyeljünk arra, hogy a kódot ne hálózati meghajtóról futtassuk, mert annak korlátozottak a biztonsági beállításai. Valamint ha hálózatról töltünk le egy DLL-t, akkor újabb Windowsok blokkolják a hozzáférést a benne lévő kódhoz. Ezt a fájl tulajdonságlapján az *Unblock* gombbal tudjuk megszüntetni.
- Többszálú alkalmazás kibővítése esetén WMI szolgáltató kiajánlani csak abból a szálból lehet, amiben használva is lesz később.

1.3 Naplózás az Enterprise Library Logging Application Block segítségével

A naplózás feladata majd minden alkalmazás készítése során előkerül. Bár meg lehet oldani saját kóddal is, érdemes valami jól konfigurálható és könnyen használható keretrendszert keresni hozzá. A segédletben a Microsoft által készített *Enterprise Library* csomag *Logging Application Block* komponensét fogjuk bemutatni. Az *Enterprise Library* újrahasznosítható, jól átgondolt komponensek gyűjteménye, mely a komplexebb szoftvereknél előkerülő tipikus feladatokra nyújt megoldást. A *Logging* blokk része egy egyszerű API-val rendelkező, sokrétűen konfigurálható naplózó megoldás.

Az *Enterprise Library* oldaláról [8] letölthetők az alkalmazásblokk könyvtárai és egy „lépésről-lépésre" útmutató a *Hand-on Labs* résznél. A hozzá tartozó *Devoper's Guide* [9] leírásban található egy jó áttekintő a *Logging* blokk használatáról. Az *Enterprise Library*hoz tartozó CodePlex oldalán [10] pedig elérhető a részletes dokumentáció (egyben, CHM formátumban is). Ezeket érdemes áttanulmányozni. A segédlet további részében bemutatjuk, hogy *ConsoleGuessGame* alkalmazásunkat hogyan lehetne naplózó funkciókkal kiegészíteni.

1.3.1 A mintaalkalmazás naplózásának megtervezése

Az alkalmazás felügyeleti modelljének elkészítése során (1.1.2 fejezet) meghatároztuk, hogy milyen eseményeket lenne érdemes naplózni, azokhoz milyen üzenetek, azonosítók és súlyossági szintek tartoznak. Itt most azt kell eldönteni, hogy ezeket milyen helyekre és milyen formában akarjuk naplózni a *Logging* blokk segítségével.

A következő általános követelményeket határozzuk meg:

R1. Egyszerűen állítható módon lehessen a naplózást ki- és bekapcsolni.

R2. Az *Error* és *Critical* súlyosságú üzeneteket szeretnénk az eseménynaplóban látni.

R3. Normál üzemmódban ezen kívül az *Information* vagy annál magasabb súlyosságú üzeneteket egy *ConsoleGuessGame.exe.log* fájlban akarjuk tárolni. A fájl CSV szerkezetű legyen, egy sorban egy naplóbejegyzés. Lehessen megkülönböztetni, hogy melyik üzenet jött az alkalmazás „üzleti" részéből, és melyik a WMI szolgáltató résztől.

R4. Igény esetén át lehessen kapcsolni, hogy a fájlba részletesebb információ is bekerüljön.

R5. Legyen lehetőség arra, hogy a WMI szolgáltató műveleteihez a *Tracing* módot is engedélyezzük. Ilyenkor a WMI-ból meghívható műveletek indítását és végét, azok összerendelését lehessen részletesen naplózni egy *GuessGameProvider.trace* fájlba.

1.3.2 Naplózás megvalósítása a Logging blokk segítségével

A *Logging* blokk segítségével történő naplózás folyamatát a következő ábra szemlélteti (6. ábra).

A *Logging* blokk használatához a következő műveleteket kell elvégeznünk.

1. *Enterprise Library* csomag telepítése NuGet segítségével

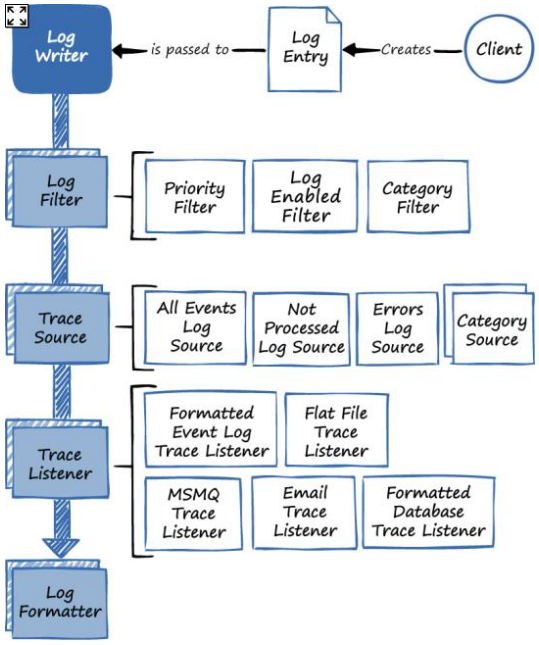
   A NuGet egy csomagkezelő, amivel .NET-es komponenseket lehet könnyen hozzáadni a projektünkhöz, majd azokat később folyamatosan frissen tartani. Aki nem használata még, annak érdemes átfutni a bevezetőjét [11].

A projekt jobb gombos menüjében a *Manage NuGet Packages…* menüpontban találjuk a NuGet felületén, itt az *Enterpise Library – Logging Application Block* nevű csomagot kell megkeresni és telepíteni.

2. Szükséges referenciák

   A NuGet csomag telepítésével a következő referenciákat adtuk hozzá a projekthez:

```
Microsoft.Practices.EnterpriseLibrary.Common
Microsoft.Practices.EnterpriseLibrary.Logging
```



6. ábra: A naplózás folyamata a Logging Application Block segítségével [9]

3. `LogWriter` példány létrehozása

   A naplózás megkezdése előtt előbb létre kell hozni egy példányt a `LogWriter` osztályból, ezt most a `Main` metódus elején tesszük meg, és a példányt eltároljuk a `Program` osztály egy statikus mezőjében.

```
Program.Logger = (new LogWriterFactory()).Create();
```

   Figyelem: ez csak az egyik módja a létrehozásnak[5].

4. Naplóbejegyzés létrehozása

   A kódban még csak a bejegyzés általános tulajdonságait mondjuk meg (szöveg, azonosító, súlyosság, stb.), az majd csak a konfigurációs állomány beállításaiból derül ki, hogy hova kerül ez a bejegyzés.

[5] Bővebben lásd az *Enterprise Library* dokumentációjának „*Creating and Referencing Enterprise Library Objects*” részét.

A naplóbejegyzéseket létrehozhatjuk a `Write()` metódus meghívásával, vagy létrehozhatunk külön egy `LogEntry` objektumot. Így az alap információkon kívül rengeteg egyéb adatot hozzáadhatunk[6] (pl. aktuális verem tartalom, hitelesítési adatok stb.).

Álljon itt egy példa a naplózó utasítás használatára (ez csak a legalapvetőbb információkat állítja be a bejegyzésen.):

```
Program.Logger.Write(message, category, priority, eventId, severity);
```

Megjegyzés: a harmadik paraméter a prioritás, ami egy egész szám lehet. A példaalkalmazásban a súlyosságot (`TraceEventType` típusú paraméter) használjuk majd, amellett a prioritásra nincs most szükség, úgyhogy ez mindenhol nulla értékű lesz. Más alkalmazásban lehet, hogy célszerűbb a prioritás vagy akár mindkettő alkalmazása. A prioritás előnye, hogy a konfigurációs fájlban egyszerűen lehet rá szűrést definiálni.

A példaalkalmazásban ezen kívül létrehoztunk egy statikus segítő metódust is, ami ezeket a gyakran használt paramétereket várja csak a naplózáshoz. Ezen kívül a metóduson belül a tényleges naplózás előtt ellenőrizzük, hogy be van-e egyáltalán kapcsolva a naplózás.

```
internal static void Log(string message, int eventId, TraceEventType severity)
```

Ezután már csak az alkalmazás kódjában a megfelelő helyeken el kell helyezni a naplózó utasításokat. A pontos szöveget, azonosítót és súlyossági szintet az alkalmazás felügyeleti modellje definiálja.

```
Program.Log(Resources.InitSuccessLogMessage,
 LogEventIds.InvalidStartupParams, TraceEventType.Information);
```

Figyeljük meg, hogy a bejegyzés szövege nincs szövegként benne a kódban, hanem azt a szerelvénybe ágyazott erőforrás-bejegyzésekből veszi (A Visual Studio IDE-ben ezt elég könnyű típusosan kezelni, lásd [11]). Továbbá itt most azt az elvet követjük, hogy az eseményazonosítót is kigyűjtjük egy konstansokat tároló struktúrába, így egy helyen látszik az összes azonosító, és azokra típusosan lehet hivatkozni.

5. Nyomkövetési információk (tracing)

   A kódban ezen kívül még el tudunk helyezni nyomkövetési (tracing) információkat. Ennek használatához egy `TraceManager` példányra lesz szükség, ez a `LogWriter` osztályhoz hasonlóan példányosítható. A nyomkövetés használata esetén egy indulás és egy vége esemény generálódik, ezeket egy közös `Guid` köti össze. Mindkettőhöz tartozik egy pontos időbélyeg, és egy metódus belépési pont.

   Célszerű `using()` blokkal együtt használni, így akkor biztos a megfelelő időpontban áll majd le. A következő egyszerű kódrészleten kívül nyomkövetés használatára látunk példát a `GuessGameProvider` kódjában vagy a dokumentációban.

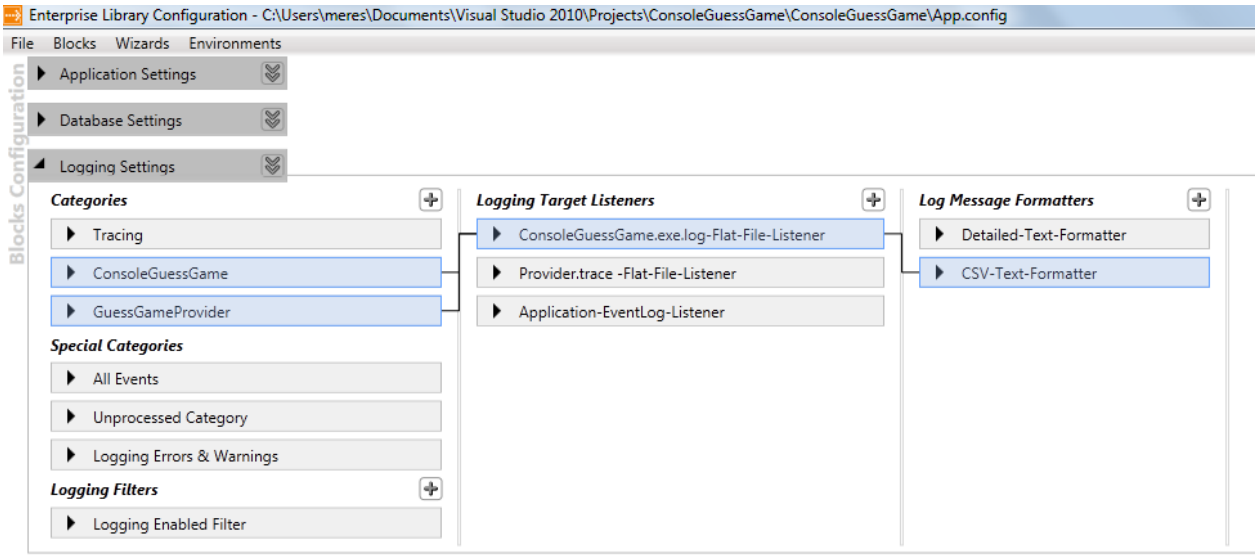
[6] Lásd a dokumentáció „*Populating a Log Message with Additional Context Information*” részét.

Ezzel készen is vagyunk a kód szintű változtatásokkal.

```
using (Program.Tracer.StartTrace(LogCategories.Tracing))
{
    if (Program.Game != null)
    {
       return Program.Game.TotalGuesses;
    }
}
```

6. Naplózási beállítások konfigurálása

Azt, hogy melyik naplóbejegyzés kerül feldolgozásra, és pontosan hova és milyen formában írjuk ki azt, teljesen egészében az alkalmazás konfigurációs állományában lévő *loggingConfiguration* rész határozza meg. Ez elég összetett lehet, szerencsére az *Enterprise Library* oldaláról letölthető és telepíthető egy jól használható grafikus szerkesztő (7. ábra). Ez elérhető külön alkalmazásként is vagy Visual Studio kiterjesztésként is.



7. ábra: Enterprise Library Configuration eszköz

Nyissuk meg az alkalmazás konfigurációs állományát (*App.config*). Ebben először a *Blocks / Add Logging Sections* paranccsal kell hozzáadni egy alap sablont. A felületnek a következő elemei vannak:

- *Logging Settings*: itt a dupla nyílra kattintva előjönnek a globális naplózási beállítások. Például itt adhatjuk meg, hogy mi legyen az alapértelmezett kategória (*Default Logging Category*) vagy, hogy engedélyezve legyen-e a nyomkövetés (*Activity Tracing Enabled*).
- *Categories*: a naplózási kategóriák. A naplózási kategória egy tetszőleges karaktersorozat lehet. Ez egy teljesen flexibilis csoportosítást biztosít, egy esemény több kategóriába is tartozhat, és egy kategóriát több helyre is kiírhatunk.

A kategória jelölheti az alkalmazás egy bizonyos részét, jelölhet adott típusú üzeneteket, egyszóval tulajdonképpen bármit. Ez a nagy szabadság elsőre kicsit meg is nehezíti a kategóriák meghatározását, érdemes mondjuk úgy nekifogni, hogy végiggondoljuk, hogy milyen bejegyzéseket szeretnénk különválogatni a többitől.

- A mintaalkalmazásban az alkalmazáshoz egy általános kategóriát definiáltunk, egyet külön a WMI szolgáltató komponensnek, és egy speciális kategóriát a nyomkövetési bejegyzéseknek.
- Az alapbeállítások között szerepel egy *General* kategória, ezt igény szerint meg lehet tartani, vagy akár el is lehet távolítani.
- Mivel a kategórianevek csak sima szövegek, érdemes nem belerakni mindenfelé a kódba őket, hanem valami konstansban definiálni ezeket. A példaalkalmazásban erre szolgál a `LogCategories` struktúra.
- Mindegyik kategóriához külön megadhatjuk, hogy milyen súlyossági szint feletti eseményeket továbbítson csak a figyelők (listener) felé.

- *Special Categories*: létezik három alapértelmezett speciális kategória. Az *All Events* kategóriába implicit belekerül az összes bejegyzés. Az *Unprocessed Category* kategóriába azok tartoznak, akiknek a kategóriája nincsen definiálva a konfigurációs fájlban. A *Logging Errors & Warnings* kategóriába pedig a naplózó rendszer saját hibái kerülnek. Ezeket nem feltétlenül kell használnunk, ha nem rendeljük őket figyelőkhöz, akkor nem írnak sehova.
- *Logging Filters*: Szűrőket adhatunk meg, melyek megakadályozhatják bizonyos bejegyzések kiírását. A *Logging Enabled Filter* egy beállítással kikapcsolja a teljes naplózást. Lehetne továbbá kategória vagy prioritás szerint szűrni, vagy saját *Logging Filter* implementálása esetén bármi egyéb alapján.
- *Logging Target Listeners*: a figyelők határozzák meg, hogy hova kerüljön az esemény. Lehet eseménynaplóba, fájlba, e-mailbe, adatbázisba, gördülő fájlba írni, de akár WMI eseményt is lehet kiváltani. Mindegyik figyelő más-más tulajdonsággal rendelkezik, pl. eseménynapló esetén meg kell adni az eseménynapló és a forrás nevét. A figyelőkhöz hozzá kell társítani egy formázót (formatter).
- *Log Message Formatters*: a formázók mondják meg, hogy az esemény melyik mezőjét milyen formában kell kiírni. Itt egy formázási karaktersorozatot kell tulajdonképpen megadnunk, ahol előre definiált tokeneket lehet használni.

Ha mindezeket beállítottuk, akkor használatba is vehetjük az alkalmazást.

7. Eseménynapló forrás beregisztrálása – Első futtatás rendszergazdaként

   Windows Vista óta nem rendszergazda jogú felhasználó nevében futtatott program csak akkor írhat az Eseménynaplóba, ha a forrás nevét először regisztráljuk. Ha ezt nem tesszük meg, akkor például a *Logging* blokk nem dob hibát az íráskor, csak nem kerül be a naplóba a bejegyzés. A regisztrálás legegyszerűbb módja a megfelelő .NET statikus metódus meghívása rendszergazdaként futó PowerShellből:

```
[System.Diagnostics.EventLog]::CreateEventSource("GuessGame", "Application")
```

Az első paraméter az eseményforrás neve, a második a napló neve. A forrás nevét a *Logging Application Block* konfigurációban az eseménynapló figyelő beállításainál tudjuk megadni, itt az alkalmazásunkra utaló nevet adjunk meg forrásnak.

A fenti utasítást ne felejtsük el berakni az alkalmazás telepítő szkriptjébe vagy legalább a telepítési leírásába.

8. Naplóbejegyzések megtekintése

   Ha elindítjuk a példaalkalmazást, találgatunk egyet, majd kilépünk, akkor az a következő naplórészletet generálja (a kimenetet megvágtuk, hogy kiférjen egy sorba).

```
2011-04-17,14:33:20,"GuessGameProvider constructor called.", Provider,107,Verbose,2112,1560
2011-04-17,14:33:20,"GuessGameProvider WMI provider created.", GuessGame,104,Verbose,2112,1560
2011-04-17,14:33:20,"ConsoleGuessGame initialized.", GuessGame,101,Information,2112,1560
2011-04-17,14:33:22,"New guess received: 45",GuessGame,103,Verbose,2112,1560
```

A részletes konfiguráció megtalálható az alkalmazás forrásában, itt most csak annyit tekintsünk át, hogy mivel teljesítjük az alfejezet elején definiált követelményeket:

R1. A naplózást a *Logging Enabled Filter* szűrő *enabled* tulajdonságával lehet teljesen kikapcsolni.

R2. Az *Application-EventLog-Listener* figyelő naplóz az eseménynaplóba, ennek a *filter* beállítása garantálja, hogy csak az *Error* és *Critical* bejegyzések kerülnek ide. Ebbe a forrásba az *All Events* speciális kategória elemei kerülnek, így biztos minden hibát elkapunk.

R3. A fő naplófájl a *ConsoleGuessGame.exe.log*, ebben a kategória alapján tudjuk megkülönböztetni az alkalmazás fő részének és a szolgáltatónak az eseményeit. A *CSV-Text-Formatter* gondoskodik arról, hogy egy bejegyzés mezői egy sorba, vesszővel elválasztva kerüljenek.

R4. A *ConsoleGuessGame.exe.log* figyelő *filter* tulajdonságánál lehet szabályozni, hogy milyen súlyosságú bejegyzések kerüljenek bele.

R5. Az *Activity Tracing* bekapcsolása esetén a *Tracing* kategóriába kerülnek a szolgáltató részletes nyomkövetési információi, ez pedig egy külön fájlba íródik ki.

A példa alapján láthattuk, hogy maga a naplózó kód hozzáadása általában egyszerű, az igazán komoly feladat a naplózandó események összegyűjtése és a naplózás szabályainak kitalálása.

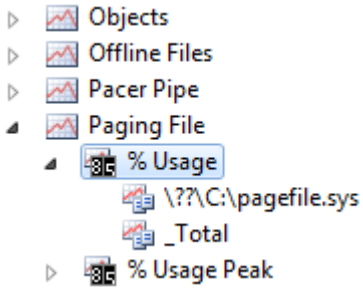
1.4 Teljesítményadatok szolgáltatása teljesítményszámlálók segítségével

Windows platformon teljesítményadatok nyilvántartására és megtekintésére szolgálnak a *teljesítményszámlálók* (performance counters). Az operációs rendszer a legtöbb komponenséhez kínál valamilyen számlálót, ezeknek a listáját és az aktuális értékét például a *Performance Monitor* (perfmon.exe) eszközben tudjunk megtekinteni.

Figyelem: bár hasonló célra használhatóak, a WMI szolgáltatók és teljesítményszámlálók egymástól független technológia. Tehát lehet például csak teljesítményszámlálókat implementálni az alkalmazásunkban WMI nélkül.

Mivel az üzemeltetők és a felügyeleti eszközök ilyen formában várják a teljesítményadatokat, célszerű a saját alkalmazásunkat is kiegészíteni, hogy teljesítményszámlálókba tudja publikálni a futási idejű jellemzőit. A .NET osztálykönyvtára kiterjedt támogatást nyújt, a legfontosabb osztályok áttekintése megtalálható az MSDN oldalán [13]. A fő osztályunk a `System.Diagnostics.PerformanceCounter`, érdemes ennek elolvasni a leírását.

A teljesítményszámlálók felépítése látható a következő ábrán (8. ábra).



8. ábra:Példa teljesítményszámlálók

A számítógépen belül a számlálókat *kategóriákba* (category) soroljuk, ilyen például a Paging File a képen. Egy adott számlálónak (pl. % Usage) több *példánya* lehet, a példányokat a nevükkel azonosítjuk.

Teljesítményszámlálók lehetnek egy- vagy *többpéldányosak* (single instance, multi instance). Az élettartamuk alapján megkülönböztetünk *helyi* (local) vagy *folyamat* (process) szintű számlálókat. A folyamat szintű számlálók megszűnnek, miután befejeződik az őket létrehozó folyamat, a helyi típusúaknak viszont megmarad az után is az értéke (csak nem frissül).

Egy számláló esetén a legfontosabb eldönteni, hogy milyen típusú legyen. Rendkívül sokféle lehetőségünk van, a főbb lehetséges fajták:

- *Pillanatnyi (instantaneous)*: mindig a legutolsó mérési értékét jeleníti meg egy egyszerű számlálónak. Például ebbe a kategóriába tartozik a `NumberOfItems64` típus.
    - Ilyen konkrét számláló a „Memory \ Available Bytes”.
- *Átlag (avarage)*: megmutatja, hogy egy művelet elvégzéséhez átlagosan mennyi idő szükséges (`AverageTimer32`), vagy mennyi elemet tudtunk az alatt feldolgozni (`AverageCount64`). Mindkét típus használatához meg kell adni egy második, `AverageBase` típusú számlálót is, ami mindig akkor növekszik eggyel, amikor befejeztünk egy műveletet.
    - Ide tartozó konkrét számláló a „PhysicalDisk \ Avg. Disk Bytes/Transfer”.
- *Ráta (rate)*: a másodpercenként végrehajtott műveletek számát adják meg az ilyen számlálók, ide tartozik például a `RateOfCountsPerSecond32` típus.
    - Példa konkrét számláló: „System \ File Read Operations/sec”.
- *Százalék (percentage)*: a számított értékeket százalékként jelenítik meg.
    - Ilyen konkrét számláló a „Paging File \ % Usage Peak”.

A konkrét számláló típusokat és azok számítási módját a `PerformanceCounterType` dokumentációjában [14] találjuk meg. Itt mindegyik főbb típushoz találunk példakódot is, ezek alapján megérthető a használatuk módja.

Miután a felügyeleti modellben definiált teljesítménymetrikákat leképeztük valamelyik teljesítményszámláló-típusra, a következő feladatunk, hogy létrehozzuk a számlálókat.

1.4.1 Teljesítményszámlálók létrehozása

Teljesítményszámlálókat létrehozhatunk kézzel vagy programozottan is [15]. Programozott létrehozáshoz a `CounterCreationData` osztályt kell használni.

A számlálók létrehozásához megfelelő jog kell, ezért ezt a feladatot érdemes elkülöníteni az alkalmazástól, és a telepítőjében végrehajtani. Ez lehet egy .NET-es program, de mivel PowerShellből könnyedén lehet .NET objektumokat létrehozni, majd azok metódusait meghívni, ezért használhatjuk azt is.

A mintaalkalmazás forrásában megtalálható egy részletes szkript, ami elvégzi a következő feladatokat (`Create-GuessGamePerformanceCounters.ps1`).

- Ellenőrzi, hogy rendszergazda futtatja-e.
- Ellenőrzi, hogy létezik-e már a létrehozandó kategória, ha igen, azt a `-Force` paraméter megadásakor le is törli.
- Létrehozza a teljesítményszámlálókat, beállítja azok nevét és leírását.
- Létrehozza a megadott kategóriát, és hozzáadja a fenti számlálókat.

Teszteléshez lehet hasznos, hogy PowerShellből a következő utasítással lehet egy teljes kategóriát és a benne lévő összes számlálót letörölni:

```
[Diagnostics.PerformanceCounterCategory]::Delete("CategoryName")
```

Figyelem: A *Performance Monitor* eszközt általában újra kell indítani, hogy észrevegye, hogy közben megváltoztattuk a számlálókat vagy azok tulajdonságait.

1.4.2 Teljesítményszámlálók értékének beállítása az alkalmazásból

A következő lépés, hogy módosítjuk az alkalmazásunk forráskódját, hogy állítsa be folyamatosan a létrehozott számlálók értékét.

A példaalkalmazásban két számlálót implementáltunk, a *TotalGuesses* és a *TotalFaultyGuesses* metrikáknak megfelelően. Itt is érdemes átgondolni, és valami konzisztens módszert kitalálni a számlálók kezelésére a kódolás megkezdése előtt. A példa egyszerűsége miatt most mindkettő számlálót a `GuessGame` osztályban helyezzük el.

1. Számláló objektumok példányosítása

   A konstruktorban létrehozzuk a számlálót reprezentáló objektumot:

```
this.totalGuessesCounter = new PerformanceCounter(Counters.CategoryName,
 "TotalGuesses", this.SessionName, false);
```

Meg kell adni a kategórianevet, a számláló nevét, a példány nevét, és azt, hogy csak olvasható legyen-e. Így a fenti utasítással egy írható példányt kapunk. Mivel a kategória neve több helyen is szerepelhet a kódban, így azt érdemes valami konstansban tárolni.

2. Számláló növelése

   A megfelelő helyen növelni kell a számláló értékét (lehet akár többel is). A példában a `Guess` metódusba került be a következő sor:

   ```
   this.totalGuessesCounter.Increment();
   ```

3. Végezetül itt most szükség van a számláló nullázására is.

   A NewRound metódusban mindkét számlálót nullázzuk.

   ```
   this.totalGuessesCounter.RawValue = 0;
   ```

4. A hibakezelést érdemes még átgondolni.

   Az alkalmazás indulásakor ellenőrzi, hogy létezik-e az alkalmazáshoz tartozó kategória. Ezen kívül a számlálókat reprezentáló objektumokat try-catch blokkban kell létrehozni.

Éles alkalmazás fejlesztése során érdemes a következőkre figyelni:

- Mivel a teljesítményadatok nyilvántartása erőforrás-igényes lehet, hasznos, ha ezt az instrumentálást ki lehet kapcsolni konfigurációs fájlból.
- Figyeljünk arra, hogy a teljesítményszámlálót reprezentáló objektum létrehozása időigényes, így az alkalmazás életciklusának elején egyszer hozzuk csak létre, és ne minden egyes módosításkor példányosítsuk.
- A teljesítményszámlálók alapvetően közös erőforrások, egyszerre többen férhetnek hozzá. Így fontos a kölcsönös kizárás biztosítása. Ezt az *Increment* és *Decrement* metódusaik biztosítják, azok belül atomi típusokat használnak. A *RawValue* tulajdonság viszont nem szálbiztos, cserébe viszont gyorsabb a módosítása.
- Gondoskodni kell az teljesítményszámlálók felszabadításakor az alkalmazás kilépésekor. Erre a *Dispose* minta [16] használata javasolt abban az osztályban, ami a teljesítményszámlálókat tartalmazza.

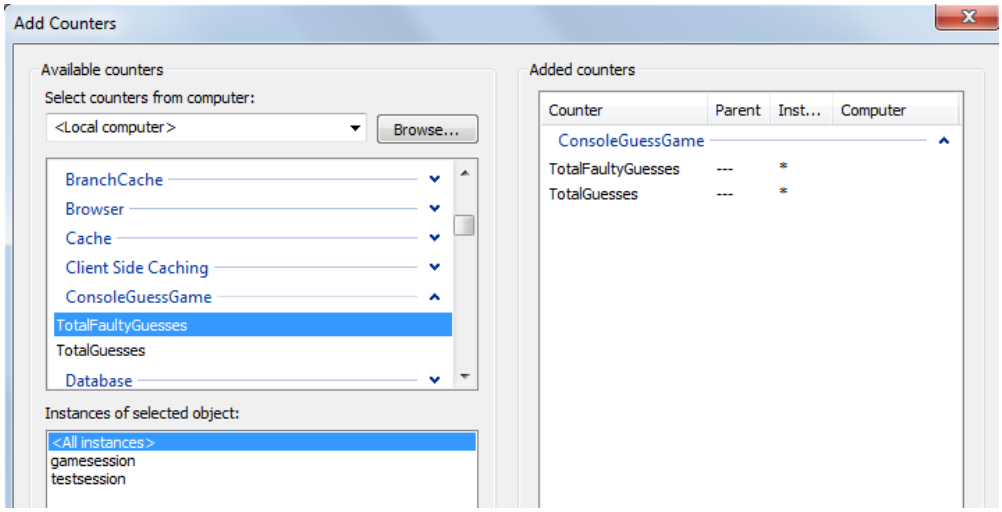
1.4.3 Teljesítményszámlálók leolvasása

Miután hozzáadtuk a szükséges instrumentálást az alkalmazáshoz, most már ki kell próbálni, hogy hogyan tudjuk leolvasni a számlálókat.

1. Indítsuk ez alkalmazásunkat, és végezzünk rajta el pár műveletet.
2. Számlálók kiválasztása

   Nyissuk meg a *Performance Monitor* (Teljesítményfigyelő) eszközt, és a menüsoron lévő zöld plusz segítségével adjunk hozzá számlálókat (9. ábra).

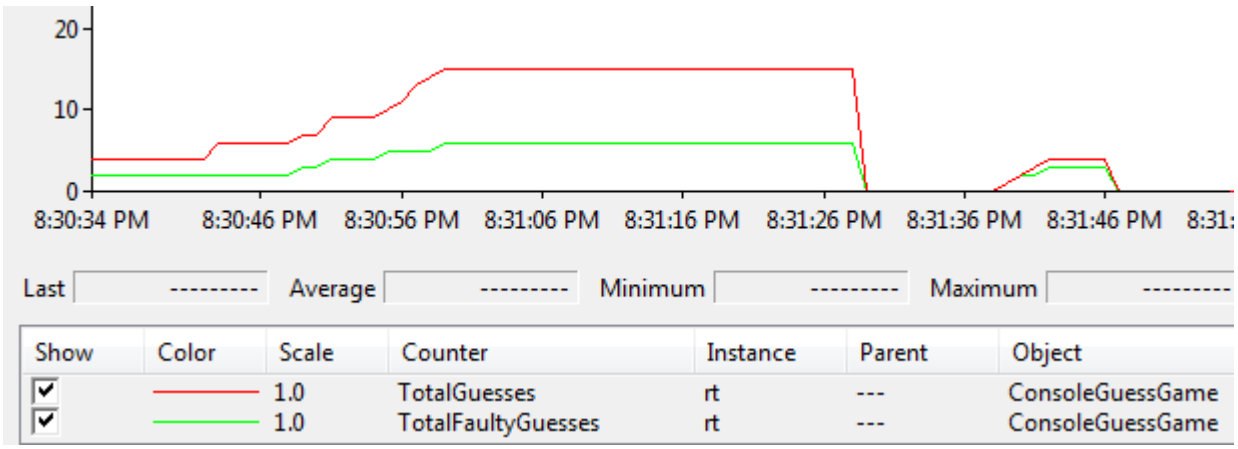
Itt ki lehet választani a kategóriához tartozó egyes számlálókat, és azt is, hogy a számláló melyik példányát akarjuk megnézni. Látszik az ábrán, hogy jelenleg két példányban futott az alkalmazásunk.



9. ábra: Számlálók hozzáadása a Performance Monitor eszközhöz

3. Adatok megtekintése

Ezek után már valós időben tudjuk figyelni a számlálók értékének alakulását (10. ábra). Az ábrán például látszik, hogy tippeltünk párat, majd utána távolról meghívtuk a `NewRound()` beavatkozó metódust WMI-on keresztül, és az lenullázta a teljesítményszámlálókat is.



10. ábra: Teljesítményadatok megtekintése

Ezzel a pár lépéssel lehet felkészíteni az alkalmazásunkat, hogy a platform teljesítményszámlálóiba is publikálják a különböző futásidejű metrikáikat. Egy újabb felügyeleti integrációs feladatot sikerült lezárnunk.

1.5 Fejlesztést segítő eszközök

A példaalkalmazás megismerése és a házi feladat elkészítése során hasznosak lesznek majd a következő eszközök.

1.5.1 .NET-es alkalmazás fordítása parancssorból

Nem szükséges ahhoz Visual Studio, hogy egy .NET-es alkalmazást vagy akár egy Visual Studioban elkészült projektet le tudjuk futtatni. A C# fordító része a .NET keretrendszernek, így egy egyszerű .NET-es programot le tudunk könnyen fordítani[7].

Erre mutat példát a példaalkalmazás letölthető változatában a `ConsoleGuessGameOriginal` program. A C# fordítót (C sharp compiler, `csc.exe`) kell csak meghívni:

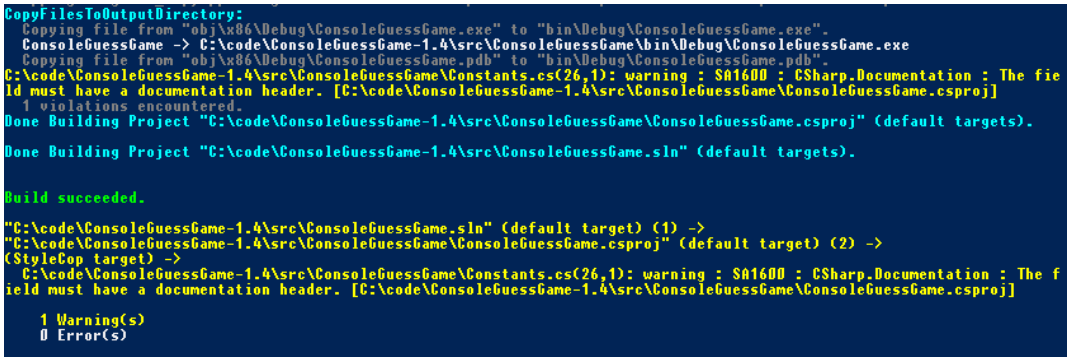
```
$NET_CSC = "C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe"
& $NET_CSC /out:ConsoleGuessGame.exe ConsoleGuessGame.cs Program.cs
```

Arra figyeljünk, hogy a megfelelő .NET keretrendszer könyvtárat adjuk meg (a fenti példa a 4.0-s verziót használja[8]).

Ha egy bonyolultabb projektünk van, amit már összeállítottunk Visual Studioban, azt is le tudjuk fordítani egy másik gépen az `MSBuild.exe` program segítségével.

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe ConsoleGuessGame.sln
```

A kimenetben szépen végigkövethetjük a teljes build folyamatot, és a végén színezve megjelennek a figyelmeztetések és hibák. A következő ábra mutat egy példát, ahol lefutott a build, azonban volt egy figyelmeztetés (11. ábra).



11. ábra: Parancssori fordítás kimenete

1.5.2 StyleCop

A fejlesztés során kritikus, hogy olvasható, szép kódot írjunk. Ebben segítenek az olyan statikus ellenőrző eszközök, amik például egy adott kódolási stílus betartására

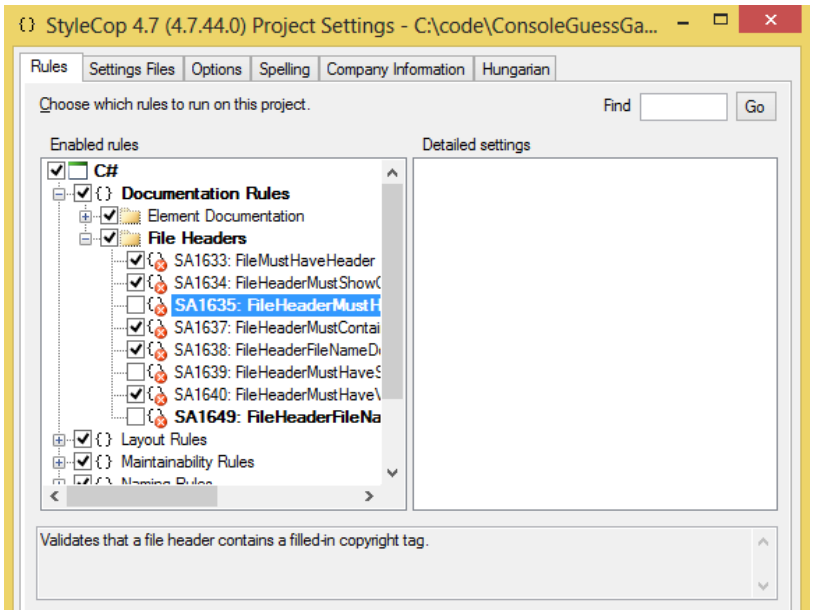
[7] Legalábbis még egy ideig, lásd Visual Studio Blog, „MSBuild is now part of Visual Studio!”, URL: http://blogs.msdn.com/b/visualstudio/archive/2013/07/24/msbuild-is-now-part-of-visual-studio.aspx
Az újabb verziókat a NuGet segítségével lehet letölteni (`nuget install Microsoft.Net.Compilers`)
[8] Ez egyben működik 4.5-ös .NET Framework esetén is, mivel az egy „in-place” frissítés a 4.0-hoz képest.

figyelmeztetnek. A házi feladat során most egy egyszerűbb eszközt, a StyleCopot fogjuk használni, ami rengeteg beépített szabállyal segíti a .NET-es konvenciók betartását.

Az ilyen kódolási stílust ellenőrző eszközöknél két dologra kell figyelni:

- Mindig lesz olyan szabály, amivel nem értünk egyet. A .NET-es világon belül is több, egymástól eltérő formázási konvenció elfogadott, mindegyik mellett és ellen is lehet érvelni. Azonban a lényeg az, hogy bármelyiket is használjuk, legyen konzisztens. Ez hatványozottan igaz, ha egy meglévő kódot egészítünk ki vagy később csatlakozunk be egy fejlesztésbe. Ilyenkor – mindegy, hogy mi szerintünk a legtökéletesebb stílus – azt kell használni, amit a meglévő kódbázis használ (vagy elkezdhetjük átírni a teljes meglévő kódot, de ez általában rengeteg munka lenne).
- Az ilyen eszközöket a fejlesztés során *folyamatosan* kell használni. Ha elkészítünk 200 sor kódot, és utána futtatjuk le az eszközt, akkor biztos, hogy legalább 50 figyelmeztetést ad majd, ezt utólag átírni nagyon kényelmetlen. Viszont ha folyamatosan figyelmeztet az eszköz, és egy-egy adott jelzés után már később úgy írjuk az új kódot, akkor nem kell sokkal több plusz energiát befektetni, és szép, konzisztens kódot kapunk majd.

A StyleCop a figyelmeztetéseit a fordító figyelmeztetéseihez hasonlóan jeleníti meg, így azokat a megszokott módon tudjuk kezelni. Ha valamelyik szabállyal nagyon nem értünk egyet, akkor megfelelő indoklással együtt ki lehet kapcsolni bizonyos ellenőrzéseket. Az ilyen kivételeket a `Settings.StyleCop` fájlban kell megadni, ehhez a StyleCop ad egy grafikus szerkesztőt is (12. ábra).



12. ábra: Stylecop szabályszerkesztő

1.5.3 Diff fájlok készítése

Ha módosítunk egy programot, akkor az egyes módosításokat úgynevezett `diff` fájlok segítségével tudjuk láthatóvá tenni. Ezek felhasználhatók később például patch-ek készítésére, a legtöbb forráskód-kezelő eszközben van ilyen támogatás. Többféle diff

formátum van, ezeket sokféle eszközzel létre lehet hozni. A `unified diff` formátumra mutat a következő részlet egy példát.

A mínusz jellel kezdődő sorokat töröltük az új fájlban, a plusz jelűeket pedig hozzáadtuk. A többi, módosítatlan sor azért szerepel, hogy be tudjuk azonosítani a módosítás helyét.

```
--- Program.cs  2012-04-21 13:17:36.000000000 +0200
+++ Program-new.cs      2013-04-12 11:12:09.749300000 +0200
@@ -74,7 +74,7 @@
                  string message = String.Format(
                      CultureInfo.InvariantCulture,
                      Resources.InvalidStartupParametersLogMessage,
-                     args == null ? "null" : String.Join(" ", args));
+                     args == null ? "null" : String.Join(",", args));
```

A fenti diff fájlt a következő paranccsal hoztuk létre:

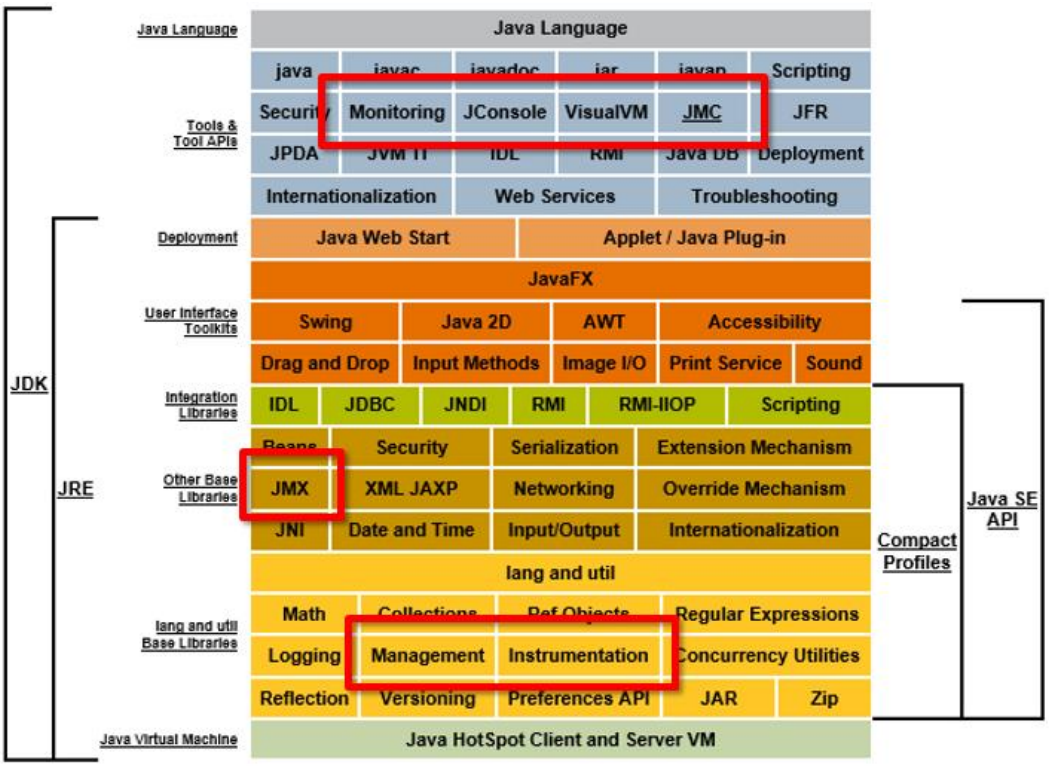
```
diff -u Program.cs Program-new.cs
```

A diff parancs megtalálható a legtöbb Unix/Linux rendszeren, Windowsra pedig például a GnuWin programcsomag segítségével telepíthető.

2 Java platform

A Java platform is messze több magánál a Java nyelvnél (lásd 13. ábra). A felügyeletre tervezés szempontjából a következő részeit érdemes kiemelni:

- *Alap osztálykönyvtárak*: naplózást és platformszintű menedzsmentinformációk elérését megvalósító osztályok.
- *Technológiák*: a Java Management Extensions (JMX) segítségével lehet saját Java alkalmazásunkat kiegészíteni menedzsment és konfigurációs adatok szolgáltatására való funkcionalitással.
- *Eszközök*: a fejlesztést, valamint a helyi vagy távoli lekérdezést támogató eszközök.



13. ábra: A Java platform felügyeleti megoldásai és eszközei [20]

2.1 Java Management Extensions

A *Java Management Extensions* (JMX) alapvetően egy instrumentációs és (távoli) menedzsment API, amelynek elemeit a `javax.management` csomagban találhatjuk.

2.1.1 Ismerkedés a JMX technológiával

Első lépésként ismerkedjünk meg a *Java Management Extensions (JMX)* platformszintű felügyeleti támogatással, illetve a „felügyeleti kliensként" alkalmazott eszközökkel. Ehhez használjunk egy kész Java alkalmazást és kapcsolódjunk hozzá lokálisan! (A távoli csatlakozás lesz a következő probléma, amivel foglalkozunk.)

1. Keressünk egy virtuális gépet, amin van JDK 8.

   Ellenőrizzük, hogy a JAVA_HOME környezeti beállítás be van-e állítva, és az a JDK könyvtárának teljes elérési útjára mutat (nem a bin-re, hanem a jdk.1.8.0-ra).

2. Itt indítsuk el valamelyik „demo" Java alkalmazást[9], mely a JDK-nak része – például a Java2Demo-t:

   ```
   C:\Program Files\Java\jdk1.8.0\demo\jfc\Java2D> java -jar Java2Demo.jar
   ```

   Linux-on az Oracle JDK telepítés után a `/usr/java/` útvonalon elindulva lesz megtalálható a demo alkalmazás.

3. Ugyanezen a gépen indítsuk el a *Java Mission Control* programot, és csatlakozzunk az előbb indított, lokálisan futó Java folyamathoz!

   A program Windows esetén a JDK *bin* könyvtárában található, *jmc.exe* néven.

   Elindítás után a bal oldali *JVM Browser* nézetben meg kell jelennie a gépen futó JVM példányoknak. Kattintsunk duplán a *Java2Demo.jar* helyi folyamatra, majd az *MBean Server* menüpontra.

4. A nyitólapon nézzük meg, hogy mennyi memóriát és CPU-t használ az adott JVM. A memória grafikonon azonosítsuk be, hogy mikor fut a garbage collector!

5. Nyissuk meg az *MBeans Browser* fület és nézzük végig a platform által alapértelmezetten rendelkezésre bocsátott platform-MBeaneket!

   Amit mindenképp érdemes megfigyelni:

   - Objektumokat látunk, amelyeket az eszközök objektumnevük alapján szerveznek egyfajta fahierarchiába.
   - Az objektumoknak vannak metódusai és attribútumai.
   - Az MBean-ekhez, attribútumaikhoz és műveleteikhez szabványos és kiterjesztett metaadatok tartoznak.
   - A Memory platform-MBean-en „hívjuk meg" a GC műveleteit és figyeljük meg hatását a heap-használatra!

   Bizonyos attribútumok írhatók is, de ezt majd a fejlesztett megoldáson lesz érdemes kipróbálni.

   Az egyes műveletek és attribútumok jelentését lásd a platform-MBeanek dokumentációjában [17].

2.1.2 Java folyamat konfigurálása távfelügyeletre

Fussuk át a platform MBeanServer, mint kiszolgáló beállításait taglaló áttekintő dokumentumot [18].

[9] Az újabb JDK-k esetén a demokat már külön fájlként lehet letölteni a JDK letöltési oldaláról.

1. Mindezek alapján a Java2Demo egy lehetséges indítási paraméterezése (a virtuális gépben), ha távolról is akarunk tudni JMX-en keresztül csatlakozni a futó folyamathoz:

   Figyelem: a következő parancsokat érdemes a Command Promptba (cmd.exe) beírni, és nem Powershell ablakból végrehajtani, mert a PowerShell megpróbálja néha értelmezni a parancsot, és nem jól adja tovább a paramétereket.

   ```
   C:\Program Files\Java\jdk1.8.0\demo\jfc\Java2D>java
   -Dcom.sun.management.jmxremote.port=9004 -Dcom.sun.management.jmxremote.ssl=false
   -Dcom.sun.management.jmxremote.authenticate=false -jar Java2Demo.jar
   ```

2. A gazdagépen elindítva a *Java Mission Control* programot a `<virtuális gép IP-címe>:9004`-es címen tudunk JMX kapcsolatot létesíteni a távoli platformkiszolgálóval.

   Persze csak akkor, ha egyik oldal tűzfalbeállításai sem szólnak közbe.

   Linux kiszolgáló esetén még egy lépést meg kell tennünk ha a virtuális gép DHCP-n keresztül kap IP címet: mivel a távoli JMX kapcsolat Linuxon csak akkor fog kiépülni, ha a `hostname -i` parancs a kiszolgáló publikus IP-címét adja vissza, módosítsuk megfelelően a `/etc/hosts` állományt[10].

   Új távoli kapcsolatot a *File / Connect… / Create a new connection* menüben tudunk létrehozni.

3. Ennél azonban nyilvánvalóan nagyobb biztonságot szeretnénk nyújtani.

   A tantárgy keretében megoldandó feladatok esetén elégséges lesz a jelszó alapú védelem. Ennek megvalósításához több út is követhető; itt nekünk elég lesz a következő „gyors és piszkos" megközelítés, amiért persze ipari környezetben jó valószínűséggel szemöldökráncolás jár. (A következők nyilvánvalóan a „kiszolgáló" oldalra vonatkoznak.)

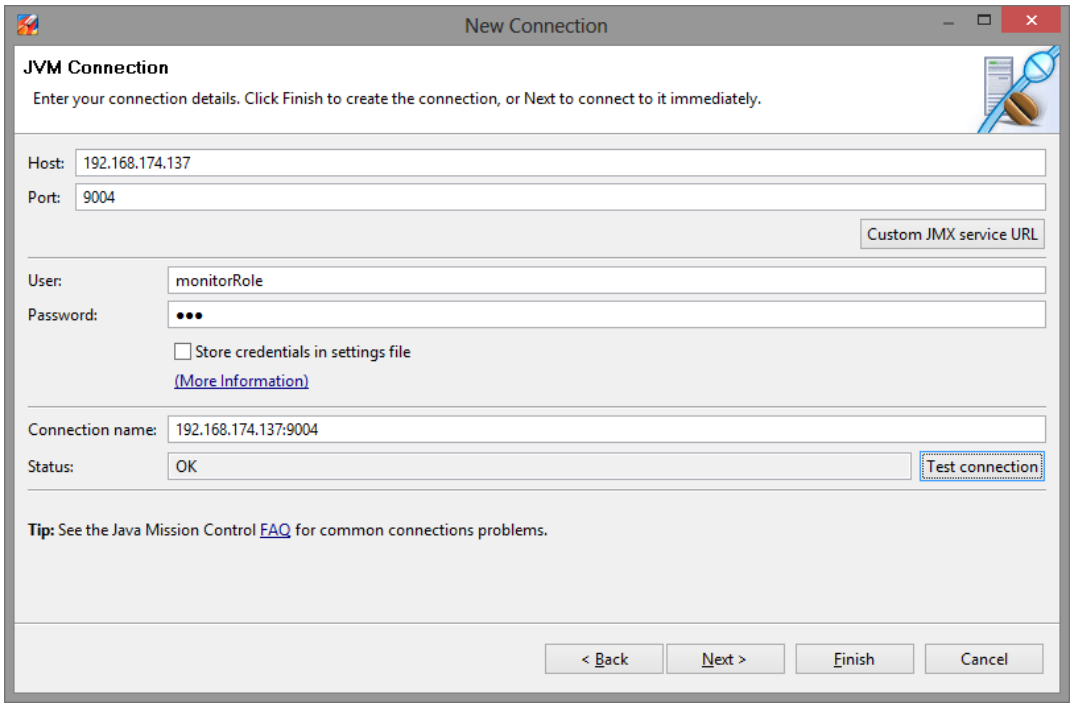
   - A `<JRE>\lib\management` könyvtárban[11] megnézzük a `jmxremote.access` állományt és úgy döntünk, hogy nekünk az alapértelmezett két szerep elég is.
   - A `jmxremote.password.template` állományról készítünk ide egy másolatot `jmxremote.password` néven, majd kikommentezzük és esetleg módosítjuk a két szerep jelszavát.
   - Windows esetén: a jelszó állománynak a tulajdonosává tesszük azt a felhasználót, akinek a nevében Java programokat futtatunk, neki „*Full control*" hozzáférést adunk a két állományon és mindenki mást megfosztunk minden jogtól.
   - Linux esetén: mindezeket a szokásos módon (*chown + chmod*) intézzük.

4. Ha most a következőképp indítjuk az alkalmazást:

[10] Lásd: http://download.oracle.com/javase/1.5.0/docs/guide/management/faq.html
[11] Ez az a JRE, amivel majd futtatjuk az alkalmazást. A kiadott virtuális gépen ez a JDK könyvtárában lévő JRE, és nem a publikus JRE.

```
C:\Program Files\Java\jdk1.8.0\demo\jfc\Java2D>java -
Dcom.sun.management.jmxremote.port=9004 -Dcom.sun.management.jmxremote.ssl=false
-jar Java2Demo.jar
```

akkor távolról a *monitorRole* név és a *jmxremote.password* állományban szereplő jelszó segítségével tudunk csatlakozni:



14. ábra. Távoli folyamathoz csatlakozás JMC-vel, szerepkört és jelszót használva

Ezen lépéseket önmagukban nem kell dokumentálni majd a feladatok megoldása során; szerepük az, hogy amikor a célalkalmazás felügyeletét valósítjuk meg, ne kelljen már azzal foglalkozni, hogy hogyan kell egy Java folyamatot távolról felügyelhetőségre paraméterezve indítani. A leadott feladatmegoldásban persze illik elhelyezni egy szkriptet, mely paraméterezve indítja a programot és a szkripben *röviden* leírni, hogy a szkript milyen előfeltételekkel él a környezettel kapcsolatban. (Közel sem tilos SSL alapú megoldást leadni, de akkor a dokumentációnak részletesebben kell tartalmaznia a szükséges lépéseket – jelszó alapú megközelítéseknél nyugodtan lehet jelen segédletre hivatkozni).

2.1.3 Alkalmazás instrumentálása MBean-ek segítségével

A következőkben azt vizsgáljuk meg, hogy hogyan tehetünk egy alkalmazást MBean-eken keresztül felügyelhetővé. Példaként tekintsük a következő egyszerű „pittyegő” programot:

```
package hu.bme.mit.ftsrg.jmxbeep;
import java.awt.Toolkit;

public class Beeper {

  public static void main(String[] args){
    while(true){
      Toolkit.getDefaultToolkit().beep();
      try {
        Thread.sleep(2000);
      } catch (InterruptedException e) {
        System.out.println("Interrupted!");
        System.exit(1);
      }
    }
  }
}
```

Az alkalmazás forráskódja letölthető a tárgy weboldaláról. A következő módon tudjuk futtatni, ha a `bin` könyvtárában állunk:

```
java hu.bme.mit.ftsrg.beep.Beeper
```

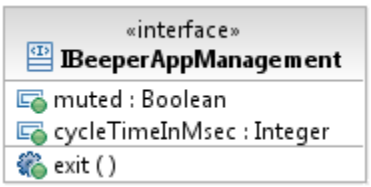
Ezt az alkalmazást nyilván nem a távfelügyeletet megcélozva tervezték. Első lépésként fogalmazzuk meg a felügyeleti célokat!

R1. Szeretnénk, ha a pittyegés távolról lehalkítható, majd visszakapcsolható lenne.

R2. Szeretnénk, ha a pittyegés gyakoriságát tudnánk befolyásolni 2 / másodperc és 0,01 / másodperc között.

R3. Szeretnénk, ha a pittyegő alkalmazást távolról terminálni lehetne.

Látható, hogy a felügyeleti igények egyben funkcionális igények is – az alkalmazás-logikát módosítanunk kell majd. Mindenesetre ezen követelmények alapján definiálhatunk is egy egyszerű logikai felügyeleti interfészt:



15. ábra. A Beeper alkalmazás logikai felügyeleti interfésze

Hogyan tehetjük alkalmazásunkat ezen interfészen keresztül menedzselhetővé a JMX technológiát alkalmazva?

A JMX technológia alapját az ún. *MBean*ek képezik; egy MBean egy menedzselt Java objektum, mely a JMX által meghatározott tervezési mintákat követi és valamely menedzselendő erőforrást reprezentál. Az MBean-ek olyan felügyeleti interfészt adnak, amely lehetővé teszi

- attribútumok írását és olvasását,
- metódusok hívását, és
- az MBean leírásának lekérdezését.

A JMX specifikáció több fajta MBean-t is definiál; mi itt a legegyszerűbb változatot, a *standard MBean*-eket mutatjuk be. A többi típus dokumentációját és alkalmazását megtalálhatjuk a technológia hivatalos segédoldaláról [19] kiindulva.

A standard MBean-ek esetén a felügyeleti interfészt egy Java interfésszel adjuk meg, melynek publikus, *get* és *set* kezdetű metódusai menedzselt attribútumot adnak meg (annak *getter* és *setter* operációi). Az interfész nevének továbbá „MBean”-re kell végződnie.

```
package hu.bme.mit.ftsrg.jmxbeep;

public interface BeeperControlMBean {
 public boolean getMuted();
 public void setMuted(boolean muted);

 public int getCycleTimeInMsec();
 public void setCycleTimeInMsec(int cycleTimeInMsec);

 public void exit();
}
```

Ezzel megadtuk, hogy az *MBeanServer* futásidőben milyen felügyeleti interfészt „ajánljon majd ki”; implementációt azonban nem rendeltünk az interfész mögé.

Itt tegyünk egy rövid alkalmazástervezési kérdéseket taglaló kitérőt. A felügyeleti kérések kiszolgálását megvalósíthatná maga a Beeper osztály – ha úgy tetszik, integrálhatnánk azt az üzleti logikával; ez azonban általában ellenjavallt. Két, fundamentálisan különböző alkalmazás-aspektusról van szó, melyeket a program jól strukturáltságának fenntartása érdekében külön szokás választani. (Ha az alkalmazáson belül ezek megfelelően szétválnak, egyszerűbb lesz például a két aspektus megvalósítását egymástól függetlenül változtatni.) A szétválasztáshoz viszont szükséges meghatároznunk/kialakítanunk a szigorúan vett alkalmazásnak egy olyan belső interfészét (vagy éppen adatszerkezeteit), mely felett a felügyeleti logika megvalósítható. Esetünkben ezt jelentheti az, hogy az alkalmazásban bevezetjük a `muted`, a `cycleTimeInMsec` és a `shouldExit` változókat:

```
public class Beeper implements Runnable{
  protected boolean muted = false;
  protected int cycleTimeInMsec = 1000;
  protected boolean shouldExit = false;

  public void run() {
    while(!shouldExit){
```

```
      if(!muted) Toolkit.getDefaultToolkit().beep();
        try {
          Thread.sleep(cycleTimeInMsec);
        } catch (InterruptedException e) {
          ...
```

Figyeljük meg azt az implicit tervezési döntést, hogy a változtatások ("menedzsment akciók") itt nem szakítják meg az alkalmazás egyes állapotaiban végrehajtott logikát. Komolyabb alkalmazások esetén azért is van szükség az állapottér-modellezésre (pl. UML állapotgépek segítségével), hogy a felügyeleti célú események és attribútum-változások hatását követhető módon tervezni tudjuk.

Most már tudjuk a deklarált felügyeleti logikát implementálni:

```
public class BeeperControl implements BeeperControlMBean{

  Beeper beep = null;

  protected BeeperControl(Beeper beep){
    this.beep = beep;
  }

  public boolean getMuted(){
    return beep.muted;
  }

  public void setMuted(boolean muted){
    beep.muted = muted;
  }
  ...
```

Statikus MBean-ek esetén szükséges, hogy az osztály neve az MBean-interfész neve legyen az „MBean" posztfix nélkül.

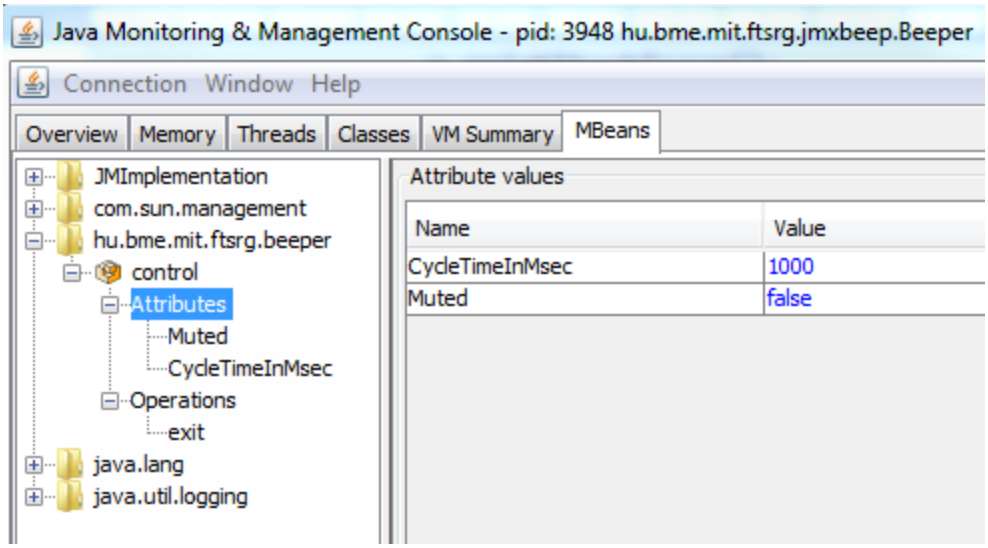
Egy dolog még hiányzik: a felügyeletet megvalósító osztályt példányosítanunk kell és objektum-névvel ellátva regisztrálni a platform-MBeanServerben. Ezt megtehetjük a `Beeper` osztály main metódusában:

```
public static void main(String[] args){

  Beeper beepbeep = new Beeper();
  new Thread(beepbeep).start();

  BeeperControl bc = new BeeperControl(beepbeep);
  ObjectName name = null;

  try {
    name = new ObjectName("hu.bme.mit.ftsrg.beeper:type=control");
  } catch (MalformedObjectNameException e) {
    ...
  }

  MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
```

```
try {
  mbs.registerMBean(bc, name);
} catch (InstanceAlreadyExistsException e) {
  ...
} catch (MBeanRegistrationException e) {
  ...
```

A kódot futtatva és a *Mission Control* eszközben (lokálisan) csatlakozva az eredmény (16. ábra):

```
java hu.bme.mit.ftsrg.jmxbeep.JMXBeeper
```



16. ábra. A kifejlesztett standard MBean JConsole-ban

A feladat megoldását előkészítendő, az itt áttekintett példaalkalmazásnak javasolt legalább egy attribútum-menedzsmentjét önállóan megvalósítani. Javasolt továbbá a már hivatkozott hivatalos JMX technológiai oldalon a „Tutorial” első és második fejezetét elolvasni.

3 Összefoglalás

A segédlet bemutatta, hogy milyen feladatokat kell megoldanunk, hogyha az alkalmazásunkat szeretnénk felkészíteni a későbbi üzemeltetésre, ha szeretnénk egy könnyen és akár automatikusan felügyelhető alkalmazást megtervezni és implementálni.

Első lépésként áttekintettük, hogy hogyan érdemes az alkalmazás felügyeletét modellezni. Ezután pedig mind a .NET, mind a Java platformhoz egy-egy alkalmazás fejlesztésén keresztül ismertettük, hogy milyen platform szintű technológiák tartoznak, és azokat hogyan is kell használni.

4 További információ

.NET

[1] MSDN Library. „WMI Provider Extensions”, http://msdn.microsoft.com/en-us/library/bb404670%28v=VS.90%29.aspx

[2] Micskei Zoltán. „Megszemélyesítési (impersonation) hiba saját WMI providerben”, http://blog.inf.mit.bme.hu/?p=243

[3] MSDN Library. „How to: Make the State of an Application Manageable”, http://msdn.microsoft.com/en-us/library/bb404677%28v=VS.90%29.aspx

[4] MSDN Library. „Security Descriptor Definition Language”, http://msdn.microsoft.com/en-us/library/aa379567%28v=vs.85%29.aspx

[5] MSDN Library. „Troubleshooting WMI Provider Extensions”, http://msdn.microsoft.com/en-us/library/bb608336.aspx

[6] MSDN Library. „Tracing WMI Activity”, http://msdn.microsoft.com/en-us/library/aa826686%28v=VS.85%29.aspx

[7] MSDN Forums. „WMI Provider (.net) Decoupled Multiple Provider instances possible?” http://social.msdn.microsoft.com/Forums/en/perfctr/thread/80e8cfc3-7b10-40df-883f-bc9f555a0a21

[8] MSDN Library. „Microsoft Enterprise Library 6” http://msdn.microsoft.com/en-us/library/dn169621.aspx

[9] Microsoft. „Enterprise Library 6 - Developer Guide”, „Chapter 5 – As Easy As Falling Off a Log”, http://msdn.microsoft.com/en-us/library/dn440731(v=pandp.60).aspx

[10] CodePlex. „patterns & practices – Enterprise Library”, http://entlib.codeplex.com/

[11] NuGet Overview, http://docs.nuget.org/docs/start-here/overview

[12] MSDN Library. „Resources Page, Project Designer”, http://msdn.microsoft.com/en-us/library/t69a74ty.aspx

[13] MSDN Library. „Performance Counter Programming Architecture”, http://msdn.microsoft.com/en-us/library/5f9bkxzf.aspx

[14] MSDN Library. „PerformanceCounterType Enumeration”, http://msdn.microsoft.com/en-us/library/system.diagnostics.performancecountertype.aspx

[15] MSDN Library. „How to: Create Custom Performance Counters”, http://msdn.microsoft.com/en-us/library/5e3s61wf.aspx

[16] MSDN Library. „Implementing a Dispose Method”, http://msdn.microsoft.com/en-us/library/fs2xkftw.aspx

Java

[17] Java Documentation. „Package java.lang.management”, http://docs.oracle.com/javase/8/docs/technotes/guides/management/index.html

[18] Java. „Monitoring and Management using JMX”, http://docs.oracle.com/javase/1.5.0/docs/guide/management/agent.html

[19] Java Documentation. Java Management Extensions (JMX), http://docs.oracle.com/javase/8/docs/technotes/guides/jmx/index.html

[20] Java Documentation. „Java SE Technologies at a Glance”, http://www.oracle.com/technetwork/java/javase/tech/index.html

5 Függelék

5.1 ConsoleGuessGame forrása a kiinduláskor

Az alkalmazásnak itt egy rövidített, megjegyzések nélküli változatát közöljük csak. A tárgy weboldalán elérhető az instrumentálással kiegészített, megjegyzésekkel ellátott kód.

5.1.1 Program.cs

```
namespace IRF.ConsoleGuessGame
{
    using System;
    using System.Diagnostics;
    using System.Globalization;
    using System.Reflection;
    using System.Resources;

    using IRF.ConsoleGuessGame.Properties;

    public static class Program
    {
        internal static GuessGame Game { get; private set; }

        public static void Main(string[] args)
        {
            // check command line arguments
            if (args == null || args.Length != 1)
            {
                Console.WriteLine(Resources.ImproperParametersUIMessage);
                Console.WriteLine(Resources.ProgramUsageUIMessage,
                         Process.GetCurrentProcess().ProcessName);

                return;
            }

            // create a new game instance and start a new round
            Game = new GuessGame(args[0]);

            while (true)
            {
                Console.WriteLine(Resources.EnterNewGuessUIMessage);

                string guessString = Console.ReadLine();

                try
                {
                    int guess = Int32.Parse(guessString);
                    GuessResult result = Game.Guess(guess);

                    switch (result)
                    {
                        case GuessResult.TooLow:
                            Console.WriteLine(Resources.TooLowUIMessage);
                            break;
```

```
                        case GuessResult.TooHigh:
                            Console.WriteLine(Resources.TooHighUIMessage);
                            break;
                        case GuessResult.Success:
                            Console.WriteLine(Resources.WinUIMessage);
                            Game.NewRound();
                            break;
                        case GuessResult.WrongInput:
                            Console.WriteLine(Resources.WrongInputUIMessage);
                            break;
                        default:
                            break;
                    }
                }
                catch (Exception)
                {
                    Console.WriteLine(Resources.InvalidNumberUIMessage);
                }
            }
        }
    }
}
```

5.1.2 GuessGame.cs

```
namespace IRF.ConsoleGuessGame
{
    using System;

    public enum GuessResult
    {
        TooLow,
        TooHigh,
        Success,
        WrongInput
    }

    internal class GuessGame
    {
        private int secretNumber;
        private Random random = new Random();

        public GuessGame(string sessionName)
        {
            this.SessionName = sessionName;
            this.NewRound();
        }

        public string SessionName { get; private set; }

        public void NewRound()
        {
            this.secretNumber = this.random.Next(100);
        }
```

```
        public GuessResult Guess(int guess)
        {
            GuessResult result = GuessResult.WrongInput;

            if (guess < 0 || guess > 100)
            {
                result = GuessResult.WrongInput;
            }
            else
            {
                if (guess < this.secretNumber)
                {
                    result = GuessResult.TooLow;
                }

                if (guess > this.secretNumber)
                {
                    result = GuessResult.TooHigh;
                }

                if (guess == this.secretNumber)
                {
                    result = GuessResult.Success;
                }
            }

            return result;
        }
    }
}
```