

**Árvai Zoltán, Csala Péter, Fár Attila Gergő, Kopacz Botond, Reiter
István, Tóth László**

Silverlight 4

**A technológia, és ami mögötte van — fejlesztőknek
HTML 5 ismertetővel bővített kiadás**



Készült a Devportal.hu közösség támogatásával

© 2010, Árvai Zoltán, Csala Péter, Fár Attila Gergő, Kopacz Botond, Reiter István, Tóth László

Lektorálta: Novák István

Minden jog fenntartva!

A könyv vagy annak bármely része, valamint a benne szereplő példák a szerzőkkel kötött megállapodás nélkül nem használhatók fel üzleti célú oktatási tevékenység során! A könyv tudásanyaga államilag finanszírozott közép- és felsőoktatásban, illetve szakmai közösségek oktatásában bármely célra felhasználható.

A szerzők a könyv írása során törekedtek arra, hogy a leírt tartalom a lehető legpontosabb és naprakész legyen. Ennek ellenére előfordulhatnak hibák, vagy bizonyos információk elavulttá válhattak.

A könyvben leírt programkódokat mindenki saját felelősségére alkalmazhatja. Javasoljuk, hogy ezeket ne éles környezetben próbálják ki. A felhasználásból eredő esetleges károkért sem a szerzők, sem a kiadó nem vonható felelősségre.

Az oldalakon előforduló márka- valamint kereskedelmi védjegyek bejegyzőjük tulajdonában állnak.

Köszönöm Adriennek végtelen türelmét és megértését.

— Árvai Zoltán

*Köszönetet mondanék Neked, kedves Olvasó, hiszen ha Te nem lennél, e
könyv se készült volna el soha 😊.*

— Csala Péter

*Köszönöm három kedves barátomnak, akik nap mint nap inspiráltak
emberfeletti szorgalmukkal és kitartásukkal, folyamatosan biztatva
szakmai utam során a kezdetektől fogva.*

— Fár Attila Gergő

*Ajánlom a könyvet Kovács Gyula és Balázs Gábor tanár uraknak, nélkülük
nem tartanék itt.*

— Reiter István

*Köszönöm apámnak, aki szerint az egyetlen dolog, amit soha nem
vehetnek el az embertől az, amit megtanult.*

— Tóth László

Tartalomjegyzék

Előszó	13
1. Silverlight és XAML alapok	15
Bevezetés a Silverlight világába	15
Mi is az a Silverlight?	15
Különböző verziók, különböző célok	16
A Silverlight felépítése.....	18
Az elmélettől a gyakorlatig	19
Dizájner-fejlesztő együttműködés, avagy út a XAML-hez	19
A Silverlight alkalmazásfejlesztés eszközei	21
A „Hello Silverlight” alkalmazás elkészítése	21
Mit kell tudni egy Silverlight alkalmazásról?	24
A projektsablonok és egy egyszerű projekt felépítése	24
Út a TestPage lekérésétől a MainPage Loaded esemény bekövetkezéséig	26
Plug-in lehetőségek	29
XAML, a felületleíró nyelv.....	31
A pixelgrafika és a vektorgrafika összehasonlítása	31
A XAML nyelv alapjai	31
A XAML nyelv szintaxisa	32
<i>További XAML lehetőségek</i>	<i>33</i>
Összefoglalás.....	33
2. Layout Management	35
Pozicionálás	35
Layout életciklus.....	36
Measure.....	36
Arrange.....	36
Bounding-Box	36
Margin és Padding	38
Layout metrika	39
Layout rounding	39
Teljesítmény	40
Virtualizáció	40
Transzformációk	41
Layout vezérlők.....	41
Grid.....	41
Canvas	43
StackPanel.....	45

WrapPanel.....	46
DockPanel.....	49
Felület kialakítása konténerek egymásba ágyazásával.....	51
Összefoglalás	55
3. Alapvető vezérlők	57
Osztályhierarchia	57
Logikai és vizuális fa.....	58
Dependency property , Attached property	59
Eseménykezelés	60
Vezérlők a gyakorlatban	61
ContentControl	62
ItemsControl	68
Egyéb vezérlők	74
Összefoglalás	77
4. fejezet: Animáció és Média	79
Alakzatok rajzolása a Silverlightban	79
Egyszerűbb geometriai formák rajzolása	79
Saját alakzatok ábrázolása	82
Vágási felületek kialakítása	83
Az alakzatok kitöltése és színezése.....	85
Képek és videók megjelenítése	87
Képek megjelenítése az Image vezérlő segítségével	87
Videó és zene lejátszása	88
Transzformációk a felületen	89
Egyszerű transzformációk	89
Saját transzformációk mátrixokkal	91
A Silverlight 3D képességei	93
Animációk a Silverlight-ban.....	93
Egyszerű animációk létrehozása	94
Átmenetek és finomítások az animációban	97
Összefoglalás	97
5. Stílusok és testreszabhatóság a Silverlightban	99
Erőforrások a Silverlightban.....	99
Az erőforrások szerepe.....	99
A StaticResource objektum	100
Az erőforrások hozzáférhetősége	100
Az erőforrások használata	100
Az erőforrás fájlok (ResourceDictionary)	101
Stílusok	103
A stílusok szerepe	103
Stílusok öröklődése	104
Implicit stílusok	104

A vezérlők testreszabása	105
A ControlTemplate-ek szerepe.....	105
ControlTemplate-ek definiálása	105
Ismerkedés a ContentPresenter-rel.....	107
A TemplateBinding fogalma	108
Vizuális állapotok	108
Vizuális állapotátmenetek	111
Komplexebb vezérlők testreszabása.....	113
Összefoglalás.....	113
6. Adatok kezelése	115
Hogyan használjuk az adatokat?	115
Különböző adatforrások, különböző adatmennyiségek.....	115
<i>Adatmegjelenítési lehetőségek</i>	<i>115</i>
A megjelenítési réteg és az adatréteg összekapcsolási módjai	116
Az adatkötés alapjai.....	116
Az adatkötés folyamata.....	116
DependencyProperty és a Silverlight Tulajdonság rendszere.....	117
Adatkötés alapjai egy egyszerű példán keresztül	118
Adatkötési lehetőségek, testreszabás.....	120
Adatkötési módok	120
A kétirányú adatkötés és az INotifyPropertyChanged kapcsolata	121
Megjelenített adat formázása, esetleges hiányosságok, hibák kezelése	123
Konverterek használata.....	124
Vezérlők közötti adatkötés	127
Listás adatmegjelenítés.....	131
A listás adatmegjelenítés alapjai	131
ObservableCollection használata mint adatforrás	132
Listás adatmegjelenítés Visual Studióban	133
Listás adatmegjelenítés Expression Blendben	136
Összefoglalás.....	139
7. Saját vezérlők készítése Silverlightban	141
UserControlok fejlesztése Silverlightban.....	141
A UserControlok szerepe	141
UserControlok létrehozása.....	142
User Controlok felhasználása	144
UserControl: Nézet vagy hagyományos vezérlő?	145
Custom Controlok fejlesztése Silverlightban	145
Egy Custom Control anatómiája.....	146
A NumericUpDownControl alapértelmezett megjelenése	147
A Value Dependency Property implementálása	148
Eseménykezelő függvények bekötése	150
Vizuális állapotok a Custom Controlban	151

Testreszabhatóság támogatása.....	153
Összefoglalás	156
8. Kommunikáció a kliens és a szerver között	157
Kommunikáció a kliens-szerver modell alapján.....	157
WCF szolgáltatások készítése, elérése és használata	158
Történeti áttekintés, a WCF fejlesztési céljai.....	158
A WCF felépítése és az ABC.....	159
Szolgáltatások létrehozása	160
Szolgáltatások elérése és használata	162
Data Transfer Object használata	165
WCF Data Services, avagy az adatok egyszerű kipublikása	167
A REST protokoll és az OData lekérdezőnyelv	167
WCF Data Services alapok.....	170
Szolgáltatások létrehozása	171
Szolgáltatások elérése és használata	173
Master-Details nézet létrehozás a WCF Data Services segítségével.....	176
WCF RIA Services, a kliens-szerver modell újragondolása	178
A Nagy Ötlet, avagy a magától adódó architektúra?	178
Szolgáltatások létrehozása	179
Szolgáltatások elérése és használata	181
Adatok módosítása és validálása WCF RIA Service-n keresztül.....	185
További kommunikációs lehetőségek röviden.....	188
HTTP alapú kommunikáció a WebClient osztály segítségével.....	188
Socket programozás Silverlightban	190
Silverlight kliensek közötti kommunikáció: Local Connection	192
Összefoglalás	194
9. A Silverlight rejtett képességei.....	195
Az alapok összeállítása.....	195
A vágólap kezelése.....	198
Drag & Drop az operációs rendszerből.....	201
Kamerakezelés	203
Teljes képernyő használata	206
Nyomtatás	207
Hálózateszlelés	210
Out-of-Browser applications	211
Kitörés	211
Testreszabás.....	213
API	215
Toast API.....	220
Fájlkezelés	221
Adatmentés és olvasás az Isolated Storage-ban	221
Beállítások mentése és olvasása Isolated Storage-ból	223

Isolated Storage Quota	224
Az igazi fájlrendszer elérése	225
My Documents	226
Kommunikáció más alkalmazásokkal	227
Office (és más COM objektumok) vezérlése	227
Kommunikáció Silverlight alkalmazások között	228
Navigation Framework.....	230
Megvalósítás egyszerű vezérlőkkel	231
Lapozások közti adatmegőrzés	232
Egyéb lehetőségek	233
Silverlight Browser Integration	234
A Silverlight plug-in	235
JavaScript hívása Silverlightből	238
HTML manipulálása Silverlightből	238
XAML manipulálása JavaScriptből	239
Összefoglalás.....	239
10. Üzleti alkalmazások fejlesztése Silverlight-ban	241
Üzleti alkalmazások fejlesztésének követelményei	241
Tesztelés és a szoftver minőség	241
Karbantarthatósági szempontok.....	241
A fejlesztő és a dizájnér közötti együttműködés	242
Architektúrális minták a prezentációs rétegben	242
A Model-View-Controller architektúrális minta	242
A Model-View-ViewModel architektúrális minta	243
A MVVM architektúrális minta gyakorlati alkalmazása.....	244
MVVM keretrendszerek használata.....	244
Az alkalmazás struktúrájának kialakítása	244
A ViewModel-ek alapfunktionalitásának implementálása.....	247
A nézetek (Views) implementálása	252
A nézetek és a ViewModel-ek összekötése	253
A ViewModel-ek és a VisualStateManager közötti kommunikáció	258
Commanding az MVVM-ben	261
Lazán csatolt metódushívások	261
A Command Pattern implementálása MVVM-ben	261
Kommunikáció a ViewModel-ek között	263
Lazán csatolt kommunikáció	263
Adatok kezelése MVVM-ben CollectionViewSource-szal	265
Az ICollectionView bevezetése a BooksViewModel-be	265
Rendezés SortDescription-ök segítségével.....	266
Szűrés a Filter delegate segítségével	267
Navigáció az adatok között	267
Adatok érvényességének ellenőrzése.....	268

Az INotifyDataErrorInfo implementálása MVVM-ben	270
Érvényességi hibák megjelenítésének testreszabhatósága	272
A ViewModel-ek Unit tesztelése	273
Unit teszt készítése a BooksViewModel-hez.....	273
Összefoglalás	275
11. Összetett adatok megjelenítése és kezelése.....	277
Táblázatos adatmegjelenítés	277
A DataGrid vezérlő alapvető használata	277
Saját oszlopsablonok létrehozása	279
Adatok csoportosítása, rendezése és szűrése.....	280
Dialógusformába rendezett adatok megjelenítése	281
Adatbeviteli felület létrehozása a DataForm vezérlővel.....	281
Lapozás az adatok között	283
Virtuális ablakok a Silverlightban.....	284
ChildWindow létrehozása és adatkötése	285
Hibaüzenetek megjelenítése ChildWindow segítségével	286
A Silverlight Toolkit.....	287
Vezérlők, melyek nem a Silverlight SDK részei	287
Összefoglalás	287
12. Moduláris alkalmazások fejlesztése	289
Lazán csatolt rendszerek	289
Architekturális megfontolások.....	289
Függőségek kezelése — Dependency Injection	289
A Managed Extensibility Framework.....	290
Exportálás	290
Importálás.....	291
Komponálás	291
Az importálás lehetőségei.....	292
Metaadatok használata.....	298
Katalógusok használata.....	300
Managed Extensibility Framework a valódi világban.....	302
A Composite Application Guidance (PRISM)	302
A PRISM által megcélzott problémák.....	303
Modulok tervezése és implementálása	303
Összefoglalás	309
13. A HTML5, az alternatív webes technológia a Silverlight mellett	311
HTML5 alapok	312
Szemantikus vezérlőelemek	312
Multimédiás vezérlőelemek	312
A vászon vezérlőelem	313
A HTML5-öt kiegészítő modulok	323
Az SVG modul.....	323

Kliens oldali adattárolás	323
A GeoLocation modul.....	324
Az Offline Web Applications modul	325
A Web Workers modul	326
A CSS 3 stílusleíró nyelv	326
Kiválasztók.....	327
A Media queries modul	329
A Colors modul	329
Dinamikus tartalomgenerálás	330
A Template layout modul (CSS sablonok).....	331
A Background and borders modul	331
A Fonts modul.....	332
A transzformációs modul (2D és 3D).....	332
Az animációs modul	332
Az áttűnés modul	333
Összefoglalás.....	333

Előszó

A web mindennapjaink részévé vált. Gyakran már fels sem tűnik, hiszen mindenütt ott van velünk, a számítógépünkön, a mobil telefonunkon, a televíziókészülékünkben, a táblagépünkön. Általában azt vesszük észre, ha éppen nem érhető el.

Webes alkalmazásokat fejleszteni már az internet gyermekkorában is kihívás volt, és még ma is az — a feladatok és az aktuális nehézségek azonban mindig is változtak. Amíg korábban egy webes alkalmazás alapvetően egy böngészőben használható honlapot, elektronikus áruházat jelentett, addig mai alkalmazásaink már kiléptek a böngésző kereteiből, és önálló életre kelve kápráztatnak el bennünket multimédiás képességeikkel — a háttérben a webet használva. A feladatok változásával együtt újabb és újabb technológiák, fejlesztőeszközök jelentek meg.

Ez a könyv azt tűzte ki célul, hogy a Microsoft Silverlight 4 technológiáját mutatja be, hozzásegítve a szakmai közösséget ahhoz, hogy magyar nyelven juthasson el hozzá az alapok elsajátítását biztosító információkhoz. A könyv nem egy erre szakosodott professzionális kiadó terméke, hanem a hazai fejlesztőközösség öt szakemberének munkájából született.

Árvai Zoltán (5., 7., 10. és 12. fejezetek), **Csala Péter** (1., 6. és 8. fejezetek), **Fár Attila Gergő** (4. és 11. fejezetek), **Reiter István** (2. és 3. fejezet) és **Tóth László** (9. fejezet) mindannyian — számos más hazai szakemberrel együtt — a Silverlight technológia ismert szakértői.

A könyv egy tömör áttekintést ad a Silverlight 4 képességeiről, segít megérteni szemléletmódját, felhasználási lehetőségeit. Nem törekszik minden részlet aprólékos elmagyarázására, inkább azokat a gyakorlati tapasztalatokat igyekszik átadni, amelyek segítségével a technológia birtokba vehető.

A könyv eredeti változatát kibővítettük a HTML 5 technológiát bemutató fejezettel, amely **Kopacz Botond** munkája (13. fejezet).

Az egyes fejezetek egymásra épülnek abban a tekintetben, hogy az első fejezettől az utolsóig folyamatosan feldolgozhatók, de azok mégsem tankönyvszerűen íródtak — inkább egy felhasználói csoport kötetlen összejövételének hangulatát hordozzák. Ez nem véletlen, a szerzők szándékosan választották ezt a megközelítési módot, hogy közvetlenebb kapcsolatot teremtsenek olvasóikkal.

A könyv és a hozzátartozó programozási mintapéldák a weben is elérhetők, a **Devportal.hu** oldalon.

Novák István

Budapest, 2011. március

1. Silverlight és XAML alapok

Ennek a fejezetnek az a célja, hogy rövid betekintést nyújtson a Silverlight csodálatos világába. Ez a bevezető egy kis ízelítőt ad a Silverlight képességeiből, illetve emellett útmutatót is kínál a további fejezetekben található egyes szolgáltatásokról. Kifejezetten azoknak szól, akik ez idáig még nem foglalkoztak ezzel a technológiával.

Bevezetés a Silverlight világába

Mi is az a Silverlight?

Erre a kérdésre hasonlóan, mint a „Mi is az a .NET keretrendszer?“, nem lehet egyszerű, rövid választ vagy definíciót adni. Általában emiatt csak körülírással szokás elmagyarázni, hogy mi is az a Silverlight. Az interneten erre eléggé sokféle megfogalmazás található, melyek között az alapvető különbség az, hogy ki melyik részét tartja fontosabbnak a Silverlightnak. Én itt két nagyjából hasonló, de tartalmilag mégis különböző „definíciót” is mutatok. Íme, az első:

„A Silverlight egy olyan cross-browser, cross-platform böngésző plug-in, amely következő generációs médiaélményt nyújt és elősegíti a gazdag felhasználói felülettel rendelkező alkalmazások (RIA) fejlesztését.”

Egy másfajta megközelítés az alábbiakat mondja:

„A Silverlight a .NET keretrendszer egy olyan cross-browser, cross-platform implementációja, amely gazdag felhasználói felülettel rendelkező alkalmazások fejlesztését teszi lehetővé, melyek lehetnek akár webesek, asztali, sőt mobil alkalmazások is.”

Tehát, a Silverlight alapjában véve egy böngésző plug-in, hasonlóan az Adobe Flash-hez. Ez azt jelenti, hogy egy Silverlight alkalmazás kódja a kliens böngészőjében fut le — alapesetben. Van arra is lehetőség, hogy Silverlight alkalmazásunk normál asztali alkalmazásként fusson. Ehhez az úgynevezett *Out-Of-Browser* szolgáltatását kell igénybe vennünk, mellyel a 10. fejezetben foglalkozunk részletesebben.

A Silverlight szakítva a .NET keretrendszer hagyományaival, amely csak és kizárólag Windows operációs rendszeren hajlandó működni, többféle platformot is támogat: Windows, Mac OSx, illetve Linux. A Linux nem a Microsoft által közvetlenül támogatott operációs rendszerek közé tartozik, de a Silverlight a *Moonlight* nevű Linux-adaptációjának köszönhetően ott is van lehetőség Silverlight alkalmazások fejlesztésére és futtatására. A Moonlight projekttel a fejezetben részletesebben is foglalkozunk.

A Silverlight nem csak több platformot, de több böngészőt is támogat! Ez azt jelenti, hogy ellentétben mondjuk egy hagyományos webes alkalmazással, ahol órákat — rosszabb esetben napokat is — el lehet tölteni azzal, hogy az alkalmazást átírjuk olyanra, hogy minden népszerű böngészőben azonos (vagy legalább közel hasonló) módon működjön, addig a Silverlight esetében erre nincs szükség. Az alkalmazást elég egyetlen böngésző alatt tesztelni és általában garantált, hogy a többiben is azonos módon fog működni.

Természetesen van néhány kivétel, például a Macnél a jobb egérgomb kérdése, ami miatt érdemes kipróbálni a többi böngésző alatt is az alkalmazást üzembe helyezés előtt.

A Silverlight az alábbi böngészőket támogatja alpból: Internet Explorer, FireFox, Opera, Chrome és a Safari. Az alábbi oldalon található egy mátrix, amely azt hivatott összefoglalni, hogy melyik operációs rendszer alatt, milyen böngésző használata esetén melyik Silverlight verzió működik:

http://en.wikipedia.org/wiki/Microsoft_Silverlight#Operating_systems_and_web_browsers.

A Silverlight a Windows Phone 7 mobil operációs rendszeren is elérhető, és az XNA mellett ennek segítségével is fejleszthetünk a platformra alkalmazásokat. (A könyvben ezzel a témával nem foglalkozunk, de akit érdekel ez a része is a Silverlightnak, annak ajánlott felkeresni az alábbi oldalt: <http://www.silverlight.net/getstarted/devices/windows-phone/>.)

Az 1-1 ábra a Silverlight X-platform és X-browser tulajdonságát szemlélteti.



1-1 ábra: A Silverlight egy cross-platform, cross-browser plug-in

A gazdag médiatámogatás vagy a következő generációs médiaélmény alatt alapjában véve azt értjük, hogy például a *Full HD* felbontású videók lejátszásához a kliens számítógépén nem kell külön feltételeznie kodeknek. Emellett idetartozik még a *Digital Rights Management* (DRM), az ún. *Smooth Streaming*, és még jó pár egyéb szolgáltatás is. A DRM segítségével jogvédett tartalmakat oszthatunk meg a világhálón, a Smooth Streamingnek köszönhetően pedig a médiafolyamot adaptív módon tudjuk letölteni, vagyis a sávszélesség függvényében rosszabb vagy jobb minőségű változatra vált át automatikusan a rendszer. Természetesen az ehhez tartozó szerveroldali támogatás is adott (*IIS Media Service* — <http://www.iis.net/download/ServeMedia>).

Végezetül a *Rich Internet Application* (RIA) támogatásról ejtenék még néhány szót. Az olyan webes alkalmazásokat, melyek az asztali alkalmazásoknál megszokott gazdag felhasználói felülettel és funkcionalitással rendelkeznek, RIA-nak nevezzük. Ez azt jelenti, hogy például két oldal közötti váltás esetén nincs villanás; aszinkron munkavégzés a jellemző — míg a háttérben dolgozik az alkalmazás, addig a felhasználói felület nem blokkolódik —; színek, formák, vezérlők, animációk gazdagsága jellemzi a felhasználói felületet.

Tehát röviden összefoglalva: *a Silverlight a Microsoft új webes interaktív médiaplatformja.*

Különböző verziók, különböző célok

Általában könyvekben ritkán szokás egy-egy adott technológia régebbi verzióiról írni, de úgy gondolom, a Silverlight ebből a szempontból kilóg a sorból, ugyanis érdemes arról is tudni, hogy az egyes verziók mögött milyen szemléletek bújtak meg.

A Silverlightnak eredetileg nem ez volt a neve, hanem WPF/E, vagyis Windows Presentation Foundation/Everywhere. Ez az elnevezés onnan jön, hogy a .NET 3.0-ban debütált új megjelenítő alrendszert WPF-nek hívják, és ez a vektorgrafikus keretrendszer annyira jól sikerült, hogy úgy gondolták,

érdemes lenne webes környezetben, böngészők alatt is elérhetővé tenni, nem csak Windows operációs rendszerben. A WPF/E a WPF-nek egy olyan leszármazott változata volt, amely a böngészőben futott.

A béta változat elkészülése után— jó microsoftos szokáshoz híven— az új technológia nevét leminősítették kódnévre, és egy sokkal frappánsabb elnevezést kerestek neki, így kapta végül a Silverlight nevet. Az 1.0-ás változat kifejezetten nagy hangsúlyt fektetett a médiatámogatásra, ezért sokan egyből a Flash vetélytársaként könyvelték el. (Mára már a két technológia meglehetősen más cél felé orientálódik.) A rendszernek volt egy nagy szépséghibája: JavaScriptből lehetett csak programozni. E technológiai korlát miatt kevés .NET fejlesztő kezdte el használni. A nem sokkal később megjelenő 1.1 alfa változatban már volt C# nyelvi támogatás. Ekkor 2007 szeptemberét írtuk.

Pár hónap múltán az 1.1 alfából lett a 2.0 béta, amellyel azt akarták jelezni, hogy a .NET nyelvi integráció igen fontos mérföldkő a Silverlight életében. A 2.0 emiatt a .NET keretrendszer nyújtotta szolgáltatások egész tárházával bővítette az 1.0 lehetőségeit. Itt jelentek meg először a vezérlők (lásd 3. fejezet), webszolgáltatás elérési lehetőségek (lásd 8. fejezet), adatmanipuláció (lásd 6. fejezet), titkosítás, stb. A programozhatóság terén is bővült a paletta, a korábban már említett C# mellett megjelentek a dinamikus nyelvek is (pl. IronPython, IronRuby).

A kilenc hónapos fejlesztési ciklusnak köszönhetően a 3.0-as változatra sem kellett sokat várni. 2009 júliusától már bárki használatba vehette az új verzió szolgáltatásait. A Microsoft azt az új koncepciót követte a Silverlight fejlesztésekor, hogy nem ők találják ki, hogy mire lenne szükségük a fejlesztőknek, hanem azt várják, hogy a fejlesztők mondják el ezt nekik. Vagyis, a felhasználói visszajelzések alapján bővítették az új kiadás funkcióit. Olyan új szolgáltatások kerültek be az új verzióba, amelyek elősegítették a Silverlightban történő üzleti alkalmazások fejlesztését. Ilyenek például a Navigáció (lásd 9. fejezet), Out of Browser mód (lásd 9. fejezet), .NET RIA Services (lásd 8. fejezet).

A <http://dotnet.uservoice.com> oldalon leadott javaslatok alapján sok-sok újítással bővült a 4.0-as változat, mind az üzleti alkalmazások terén például nyomtatás (lásd 9. fejezet), MVVM támogatás (10. fejezet), MEF (12. fejezet) mind pedig a média terén például mikrofon és webkamera támogatás (lásd 9. fejezet), H.264 codec támogatás. Az új verzió emellett sok régebbi megkötést is feloldott. Például a 4.0-tól kezdve már használható a teljes billentyűzet a teljes képernyős módban is, az Out-Of-Browser módban futó alkalmazások végezhetnek COM hívásokat, belekerültek végre is a rendszerbe a fájlok megnyitásához és mentéséhez kapcsolódó dialógusablakok, és így tovább. Tehát, összességében elmondható, hogy a Silverlight 4.0 már egy egészen jól használható rendszerré nőtte ki magát, leküzdve a kezdeti gyerekkori betegségeit.

A Silverlight következő változatában olyan újítások várhatóak, mint például a perspektivikus 3D lecserélése teljes 3D támogatásra, GPU támogatás videó dekódolásnál, 64 bites böngészők támogatása, stb.

A Silverlight különböző verzióinak tárgyalása esetén nem mehetünk el a Moonlight mellett anélkül, hogy nem ejtenénk róla néhány szót. A Moonlightot (melynek logója az 1-2 ábrán látható) ugyanaz a csapat fejleszti, mint a C# Linux-os változatát, vagyis a Mono-t (ami spanyolul majmot jelent). Sajnos, itt túl sok jó hírrel nem szolgálhatok, ugyanis eléggé nagy lemaradásban vannak a srácok. A könyv írásának idején a MoonLight még csak a Silverlight 2.0 funkcionalitásának egy részét valósította meg. Emiatt igazándiból csak a Silverlight árnyékának tekinthető.

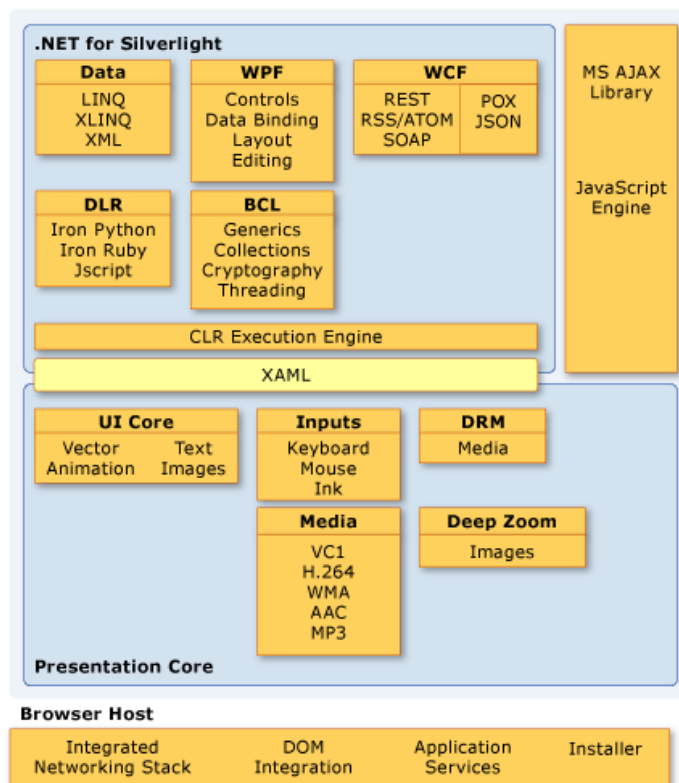


1-2 ábra: Moonlight (shadow)

A Silverlight felépítése

A Silverlight architektúráját három nagyobb részre lehet felosztani, amelyeket az 1-3 ábra szemléltet:

- Alap megjelenítő rendszer (*Presentation Core*)
- Silverlightos .NET Keretrendszer (*.NET for Silverlight*)
- Telepítő és automatikus frissítő rendszer (*Installer*)



1-3 ábra: A Silverlight architektúrája

A megjelenítő rendszer minden olyan dolgot magába foglal, amely a felhasználói felülettel kapcsolatos, kezdve a vektor-grafikus alakzatoktól (4. fejezet), a vezérlőkön (3. fejezet) és azok elrendezésén (2. fejezet) át egész az adatkötésig (ez kapcsolja össze az adatréteget a megjelenítési réteggel, amint azt a 6. fejezet tárgyalja). Ide értjük még a különböző input eszközök (pl. egér, billentyűzet) és médiák kezelését is.

A Presentation Core-hoz még hozzátartozik a *DRM* (jogvédtett médiák kezelésére szolgáló technika) és a *DeepZoom* (gigapixeles képek vagy képkompozíciók gyors megjelenítésére alkalmas technológia) is, de ezekkel ebben a könyvben nem foglalkozunk. Viszont érdemes az alábbi két oldalon utánukolvasni:

- <http://msdn.microsoft.com/en-us/library/cc838192.aspx>
- <http://www.microsoft.com/silverlight/deep-zoom/>

A Silverlight .NET keretrendszere a Windows-os .NET keretrendszer egy rendkívül lekarcsúsított változata. Ez azt jelenti, hogy az olyan dolgokat, amelyeknek nincs értelme ebben a környezetben, egyszerűen kihagyták ebből a változatból (például az ADO.NET nagy részét), illetve ahol sok alternatíva közül lehetett válogatni, ott csak néhányat hagytak meg közülük (például kriptográfiai osztályok). Ezen kívül a CLR-t (a .NET-es kódok futtató és felügyelő környezetét) is átszabták, és így egy új úgynevezett „lightweight” (pehelysúlyú) CLR kapott helyet a Silverlightban. Mind az alapkönyvtárak, mind a CLR, mind pedig a rájuk

épülő szolgáltatások karcsúsítása oda vezetett, hogy a Silverlight telepítőkészletét sikerült 5 megabyte körüli méreten tartani.

Ahogy az 1-3 ábra is mutatja, az alap megjelenítő rendszer és a Silverlight-os .NET keretrendszer között elhelyezkedik egy XAML nevezetű komponens, melyet a felületek leírására használunk. Ez teremti meg a kapcsolatot a két világ között.

Az 1-3 ábra kapcsán utolsó előtti dologként még érdemes kiemelni, a legalján látható *Browser Host* lehetőséget. A Silverlight ugyebár a böngészőben fut. Ez azt eredményezi, hogy bizonyos műveletekhez van joga, másokhoz viszont nincs. Erre a szakirodalomban azt szokták mondani, hogy az alkalmazás úgynevezett *sandbox*-ban (magyarul a „homokozó” fordítása terjedt el) fut. A homokozón belül bármit megtehet (persze amit a szülei megengednek neki), de azon kívül már meglehetősen korlátozottak a lehetőségei. A homokozón belüli képességek közé tartozik egy igen érdekes szolgáltatás is, mégpedig a Silverlight alkalmazást tartalmazó weboldal .NET kódból történő elérése és manipulálása.

A Silverlight alkalmazásokat nem lehet egy az egyben közvetlenül használni, hanem be kell őket ágyazni egy weboldalba, mint a Flash-es alkalmazásokat vagy a Java-s applet-eket.

Ezt a szolgáltatást a Silverlight terminológiában *HTML Bridge*-nek nevezik. Ennek segítségével arra is van lehetőség, hogy egy JavaScript függvényt meghívjunk C#-ból vagy éppenséggel fordítva. (Az HTML Bridge-dzsel a 9. fejezetben foglalkozunk részletesebben.)

Végezetül vizsgáljuk meg a Silverlight telepítő komponensét, mely több szempontból is érdekes! Egyrészt a már korábban is említett 5 megabyte körüli mérete miatt, másrészt pedig a kb. 10 másodperces telepítési idő miatt. Ezen felül még érdemes azt is megjegyezni, hogy a telepítés után nem kell újraindítani a böngészőt, hanem egyszerűen csak frissíteni kell azt az oldalt, amelyen a Silverlight tartalom található. Többféle telepítési módja is létezik, melyeket részletesen az *Eszközök* című részben tárgyalunk.

Az elmélettől a gyakorlatig

Dizájner-fejlesztő együttműködés, avagy út a XAML-hez

Ahhoz, hogy egy alkalmazás hasznos legyen, nem elegendő az előírt funkcionalitás megvalósítása, annak használhatónak is kell lennie. Ehhez arra van szükség, hogy a programozó és a felhasználó elképzelései közötti hatalmas szakadékot valaki betömje, vagyis az adatokból érthető információt állítson elő, a szolgáltatásokat pedig könnyen, intuitívan használható módon elérhetővé tegye. Ez a nemes feladat a dizájnerekre hárul.

A problémát tovább fokozza az a tény, hogy a fejlesztők és a dizájnerek között is van némi űr. Nézzük meg a problémát fejlesztői szemmel egy példán keresztül! Megbízunk egy dizájnerrel azzal, hogy készítsen XY cég új weboldalához egy arculati tervet. A dizájner el is végzi a munkáját, de mit kapunk eredményül? Egy képet, és hozzá pár sornyi kommentet, hogy a balmargó 20px, a padding pedig 5. „Hát ezzel most jól ki lettünk segítve.” — mondhatnánk. Persze, vannak olyan dizájnerek is, akik ennél kicsit segítőkészebbek: tőlük is egy képet kapunk, viszont több rétegre szétbontva (egy PSD állományban). Ezek szétvagdosása ellenben már a mi feladatunk. Nagy ritkán találkozni olyan dizájnerekkel, akik az általuk elképzelt kinézeti tervet képesek áttenni egy HTML oldalba — illetve a hozzátartozó CSS fájlba. Az oldal ilyen tartalmilag üres változatát szokás *mockup*-nak is hívni. Ez már majdnem jó, de még ezt is át kell transzformálni például ASP.NET-ben használható formájúvá. Mindemellett itt is igaz az, hogy a vezérlők testreszabhatósága gátat szabhat a dizájner elképzeléseinek.

A probléma abban gyökeredzik, hogy a fejlesztő és a dizájner nem egy nyelvet beszélnek, vagyis a dizájner és a fejlesztő is a saját eszközeit és környezetét használja arra, hogy megalkossa a maga részét. A Silverlight ebben nyújt újat azáltal, hogy egy közös nyelvet biztosít, melyet mindketten ismernek és ad egy tervezőeszközt is, melyet mindketten képesek használni. Az előbbi XAML-nek hívják, az utóbbit pedig Expression Blendnek.

Ezt a megoldást úgy sikerült elérnie a Microsoftnak, hogy az üzleti logikát különválasztotta a megjelenítéstől. Így a dizájner mindaddig dolgozhat a megjelenésen (Expression Blendben), amíg a

fejlesztő a funkcionalitás implementálásával van elfoglalva (Visual Studióban), és teszik mindezt egyetlen projektben. A WPF bevezetésekor az alábbi jelige kezdett el terjedni ennek a filozófiának a rövid és tömör megfogalmazásaként:

Application = Markup + Code

Vagyis, hogy egy alkalmazás két főrészből áll, egy felület leíró és egy kód részből, lásd 1-4 ábra. Ez a fajta szeparáció elősegíti a dizájnerek és a fejlesztők közötti hatékony együttműködést.

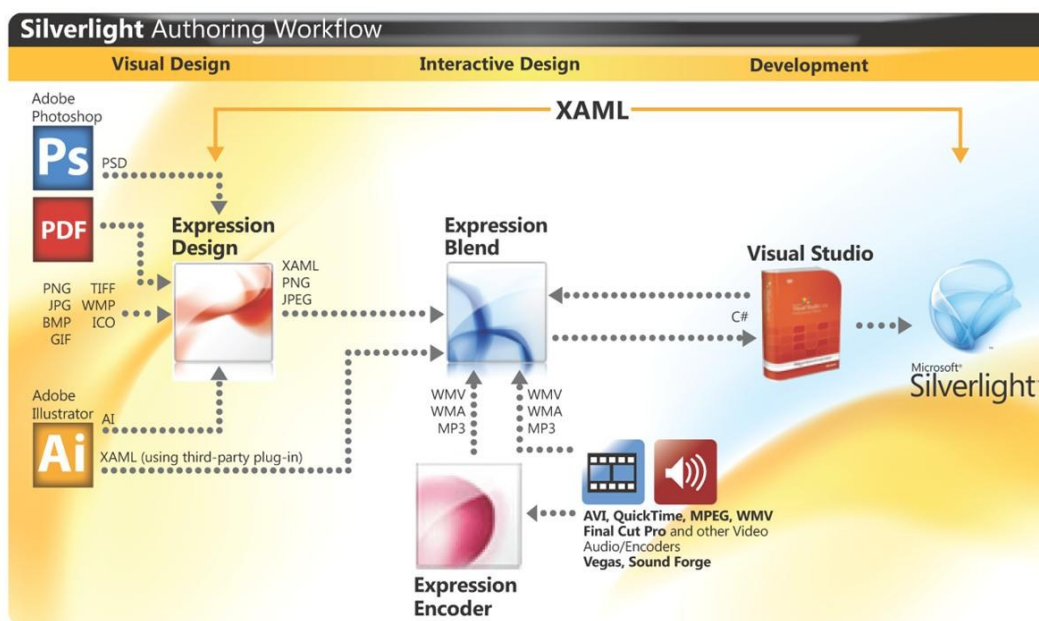


1-4 ábra: Kinézet + Funkcionalitás ~ = Alkalmazás

Ajánlott olvasmány ehhez a témakörhöz a *The New Iteration* nevezetű whitepaper:
<http://windowsclient.net/wpf/white-papers/thenewiteration.aspx>.

Az *Expression Blend*, amelyben az alkalmazások felületét lehet szerkeszteni, az Expression termékcsalád részét képezi, ahol sok más olyan szoftver is megtalálható, melyek segítséget jelentenek a Silverlight és WPF alkalmazások fejlesztésében.

Elsőként ott van az *Expression Designer*, mely pixel- és vektorgrafikus alakzatok szerkesztésére alkalmas. Egy másik fontos termék az *Expression Encoder*, mellyel a Silverlight nyújtotta médiatámogatást tehetjük még gazdagabbá. Egy Silverlight alkalmazás fejlesztésénél célszerű mindent bevetnünk a minél jobb *felhasználói élmény* érdekében, ezért egy alkalmazás elkészítésekor általában több eszközre is szükségünk van eme nemes cél eléréséhez, mint ahogyan azt az 1-5 ábra is mutatja.



1-5 ábra: A Silverlight alkalmazásfejlesztés eszközei

A Silverlight alkalmazásfejlesztés eszközei

Mielőtt megvizsgálánk, hogy milyen eszközöket, programokat kell feltelepítenünk a Silverlightos alkalmazások fejlesztéséhez, nézzük meg, mire van szüksége a felhasználónak!

A kliensnek mindösszesen csak a Silverlight megfelelő verziójához tartozó run-time-ot kell feltelepítenie, amelyet kétféle módon is megtehet. Az egyik lehetőség, hogy ellátogat az alábbi oldalra, és letölti a mindenkor éppen aktuális változatot: <http://go.microsoft.com/fwlink/?LinkID=149156>. A másik lehetőség, hogy az alkalmazás első használatbavétele előtt telepíti fel. Ez oly módon történik, hogy a Silverlight alkalmazást hosztoló weboldal lekérdezi, hogy milyen Silverlight verzió van feltelepítve a kliensnél, és ha nem megfelelő, akkor egy képen keresztül tájékoztatja erről, amelyre rákattintva a felhasználó eljuthat az előbb említett oldalra. Az 1-6 ábrán ez a kép látható, amelyet természetesen le is lehet cserélni — sőt kifejezetten ajánlott, de erről majd kicsit később.



1-6 ábra: A Silverlight nincs telepítve

A felhasználói oldal után térjünk vissza a fejlesztői oldalra! Egy Silverlight fejlesztőnek az alábbi három szoftvert kell mindenféleképpen feltelepítenie ahhoz, hogy hatékonyan és kényelmesen tudjon fejleszteni:

- Visual Studio 2010
- Silverlight 4 Tools for Visual Studio 2010
- Expression Blend 4

A Silverlight 4 Tools for Visual Studio 2010 az alábbi oldalról tölthető le:

<http://go.microsoft.com/fwlink/?LinkID=177428>. Ez valójában egy szoftvercsomag, amelyben olyan dolgok kaptak helyet, mint például a Silverlighthoz tartozó Visual Studio projektsablonok, a Silverlight SDK vagy a Silverlight fejlesztői futtatókörnyezete (amely hibakeresési támogatást nyújt), stb.

Az Expression Blend 4 60 napos demó változata önmagában nem érhető el, hanem az Expression termékcsaládot tartalmazó Expression Studio-val együtt szerezhető csak be az internetről, az alábbi weboldaltól: <http://expression.microsoft.com/en-us/cc507094.aspx>.

Van arra is lehetőség, hogy a Visual Studio helyett például Eclipse-t használjunk, mint IDE-t, így nem csak Windowsos környezetben fejleszthetünk Silverlight alkalmazásokat. Ebben az esetben az Eclipse4SL kiegészítésre lesz szükségünk, mely az alábbi oldalról szerezhető be: <http://www.eclipse4sl.org/>.

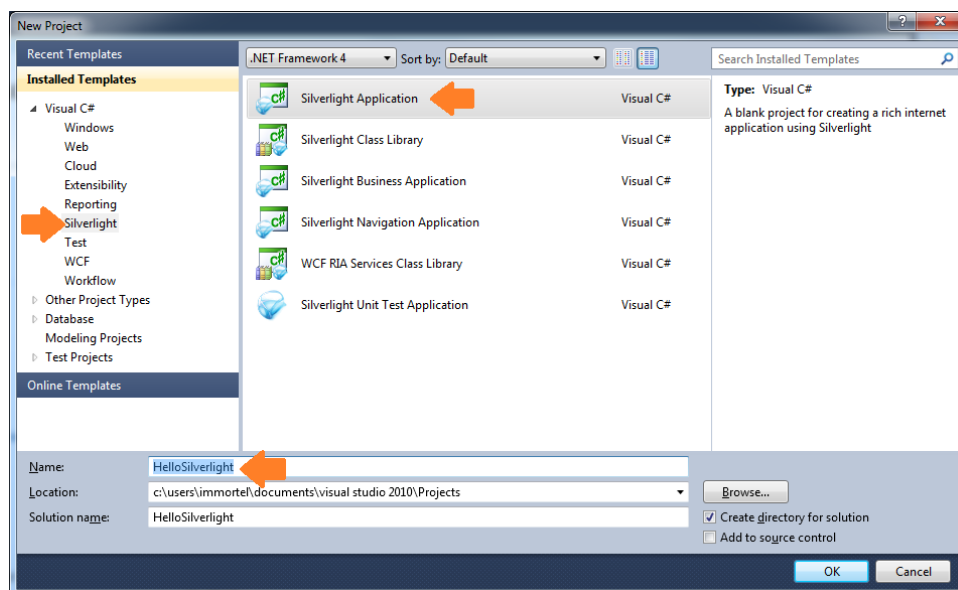
Arról is érdemes említést tenni, hogy Silverlight alkalmazások hosztolásához semmilyen extra dolgot nem kell feltelepítenünk a szerveroldalra (tehát .NET keretrendszer sem), mindössze csak a MIME type-ok közé fel kell vennünk a .xap kiterjesztést az alábbi típussal: **application/x-silverlight-app**.

A „Hello Silverlight” alkalmazás elkészítése

Ebben a részben lépésről lépésre — minimális magyarázattal ellátva — megmutatom, hogy miként lehet elkészíteni a „Hello World” alkalmazást Silverlightban. A következő alfejezetben pedig részletesen el fogom magyarázni azt is, hogy mit kell tudni egy Silverlight alkalmazásról.

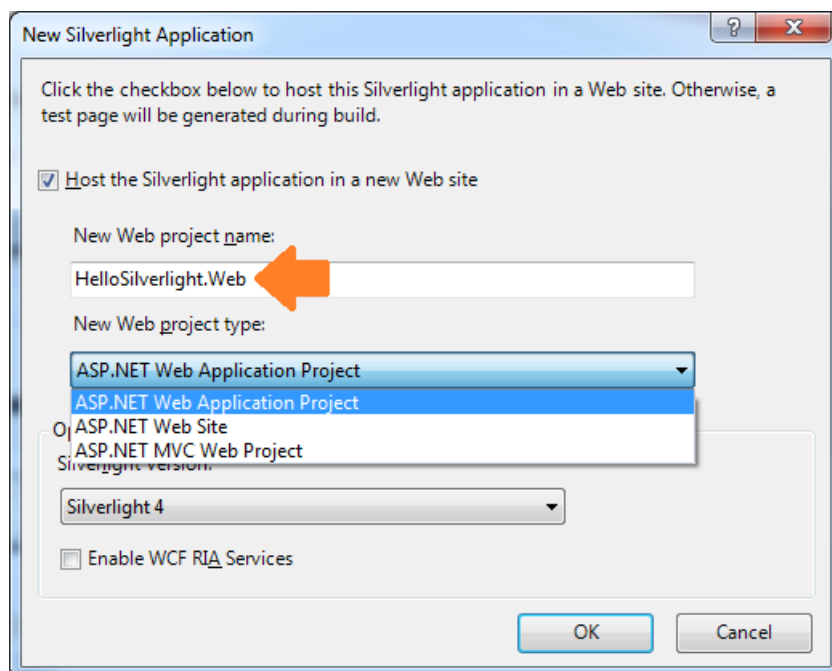
1. Silverlight és XAML alapok

Legelőször is, telepítsük fel az előző részben felsorolt alkalmazásokat, majd ha ezzel megvagyunk, akkor indítsuk el a Visual Studio 2010-et! Itt válasszuk a File ➤ New Project menüpontot, és keressük meg a felugró ablak bal oldalán a projektsablon kategóriák között a Silverlightot! Válasszuk ki a *Silverlight Application* elnevezésű sablont, és adjuk a projektnek a **HelloSilverlight** nevet, mint ahogyan azt az 1-7 ábra is szemlélteti!



1-7 ábra: HelloSilverlight projekt létrehozása

Miután rákattintottunk az OK gombra, az 1-8 ábrán látható dialógusablak tárul elénk, mely arról informál bennünket, hogy a Silverlightos alkalmazás önmagában nem futtatható. Továbbá megkérdezi, hogy létrehozzon-e neki egy teljes ASP.NET hoszt projektet, vagy bőven elegendő számunkra egy dinamikusan generált teszt HTML oldal is. Válasszuk az alapbeállítást, vagyis azt, hogy egy ASP.NET weboldal hosztolja az alkalmazásunkat!



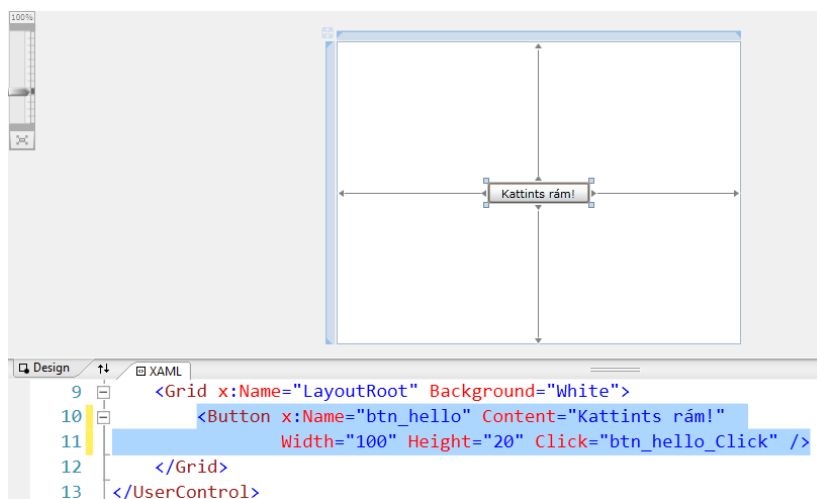
1-8 ábra: A Silverlight alkalmazást egy ASP.NET webalkalmazás hosztolja

Az OK gombra történő kattintás után a Visual Studio legenerálja nekünk a két projektet, és megnyitja a **MainPage.xaml** fájlt, amely az alkalmazásunk főoldalának megjelenését írja le. Ebben az ASP.NET-hez hasonlóan, deklaratív módon tudjuk leírni a felhasználói felületet XML — pontosabban XAML — alapokon.

Alapértelmezett módon a Visual Studio osztott képernyős módban nyitja meg a XAML fájlt, ahol felül a tervező nézet látható, alul pedig a felületet leíró kód. A kódban keressük meg a **Grid** elem nyitó és záró tagját, és a kettő közé helyezzük el az alábbi kódot:

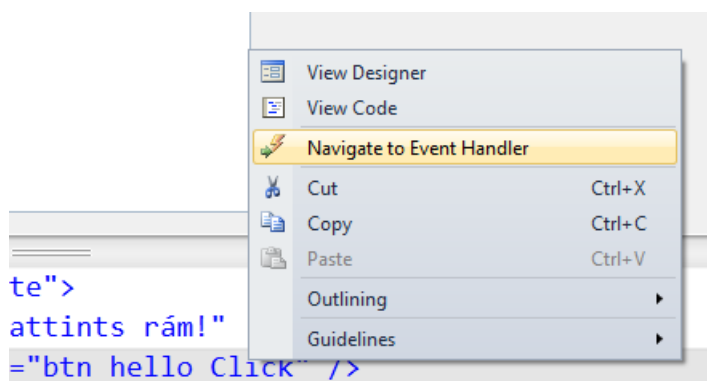
```
...
<Grid x:Name="LayoutRoot" Background="White">
    <Button x:Name="btn_hello" Content="Kattints rám!" Width="100" Height="20"
        Click="btn_hello_Click" />
</Grid>
```

Ezzel a kódrészlettel egy nyomógombot hoztunk létre, amely **btn_hello** névre hallgat, a felirata az, hogy „Kattints rám!”, és 100 egység széles, illetve 20 egység magas. Mindemellett pedig feliratkoztunk a kattintás eseményre (**Click**). Az 1-9 ábra szemlélteti ezt a lépést.



1-9 ábra: Egy gomb elhelyezése a felhasználói felületen

Mentsük el a fájlt, majd a **btn_hello_Click** szövegbe kattintsunk bele, és jobb egérgombot lenyomva a helyi menüből válasszuk ki a *Navigate to Event Handler* parancsot, ahogyan azt az 1-10 ábra is mutatja!



1-10 ábra: Ugrás az eseménykezelő függvényhez

Helyezzük el az alábbi kódrészletet az eseménykezelő függvényben, mely majd futásidőben lecseréli a gomb szövegét „Hello Silverlight”-ra:

```
btn_hello.Content = "Hello Silverlight!";
```

Futtassuk az alkalmazást, majd pedig kattintsunk rá a gombra! Ezt a lépést láthatjuk az 1-11 ábrán.



1-11 ábra: A *HelloSilverLight* alkalmazás működés közben

Mit kell tudni egy Silverlight alkalmazásról?

A projektsablonok és egy egyszerű projekt felépítése

Amikor a Visual Studióban létre akarunk hozni egy új Silverlight alkalmazást, akkor sok projektsablon közül válogathatunk (lásd 1-7 ábra). Az előző részben a lehető legegyszerűbbet használtuk, a *Silverlight Application* sablont, most nézzük meg, mire való a többi!

- A *Silverlight Class Library* segítségével Silverlightos osztálykönyvtárakat hozhatunk létre. Fontos megjegyezni, hogy Silverlight alatt csak ilyen típusú osztálykönyvtárakat használhatunk, normál .NET Class Library típusú projekteket vagy lefordított dll-eket nem!
- A *Silverlight Business Application* projektsablon segítségével olyan alkalmazást hozhatunk létre, amely *WCF RIA Services*-t használ (erről bővebben a 8. fejezetben) és ennek megfelelően sok előredefiniált hasznos elemet tartalmaz.
- A *Silverlight Navigation Application* olyan sablon, amelyben a képernyők közötti navigáció engedélyezve van. Azért van két különböző projektsablon (navigációt támogató és nem támogató), mivel nem minden feladat megoldásához van szükség navigációra, sok Silverlight alkalmazás egyetlen képernyőből áll csak. A navigációhoz szükséges könyvtárak (dll-ek) nem képezik az alap Silverlight run-time részét, ezért azokat az alkalmazás mellé kell csomagolni, mert ha nem használjuk őket, akkor csak feleslegesen növelik az alkalmazás méretet.
- Az 1-7 ábrán látható utolsó projektsablon, vagyis a *Silverlight Unit Test Application* nem a Silverlight 4 Tools for Visual Studio 2010-zel települ fel, ugyanis ez a Silverlight Toolkitben található (mellyel a 11. fejezetben fogunk részletesebben foglalkozni). A Silverlight alkalmazások tesztelésénél használt egység tesztekkel (Unit Test) pedig a 10. fejezet foglalkozik bővebben.

Ezek után nézzük meg, hogyan épül fel egy egyszerű Silverlight alkalmazás! Ehhez az előzőekben elkészített **HelloSilverlight**-ot fogjuk használni. Ha megnézzük a Visual Studióban a Solution Explorer-t, akkor láthatjuk, hogy a Silverlight alkalmazást hosztoló ASP.NET webalkalmazás neve **HelloSilverlight.Web** lett. Alapból az a Silverlight projektneve + „.Web” azonosítót kapja, de ez átírható, ahogyan az 1-8 ábrán is látható a New Silverlight Application ablakban.

Egy standard ASP.NET-es alkalmazáshoz képest itt 3 fájlt és egy új mappát is kapunk. A három fájl közül kettő arra való, hogy a Silverlightos alkalmazást ki tudjuk próbálni. Ezek a **TestPage**-re végződő fájlok, amelyek közül az egyik egy sima HTML, a másik pedig egy ASP.NET oldal. A harmadik fájlnk a **Silverlight.js** névre hallgat, mely sok-sok hasznos JavaScript kódot definiál, amelyekről a *Plug-in lehetőségek* című részben lesz szó. A tesztoldalak tartalmával a következő részben fogunk foglalkozni. Az új mappánkba, a **ClientBin**-be a lefordított Silverlightos alkalmazás kódja kerül, mint ahogyan azt a mappa neve is sugallja.

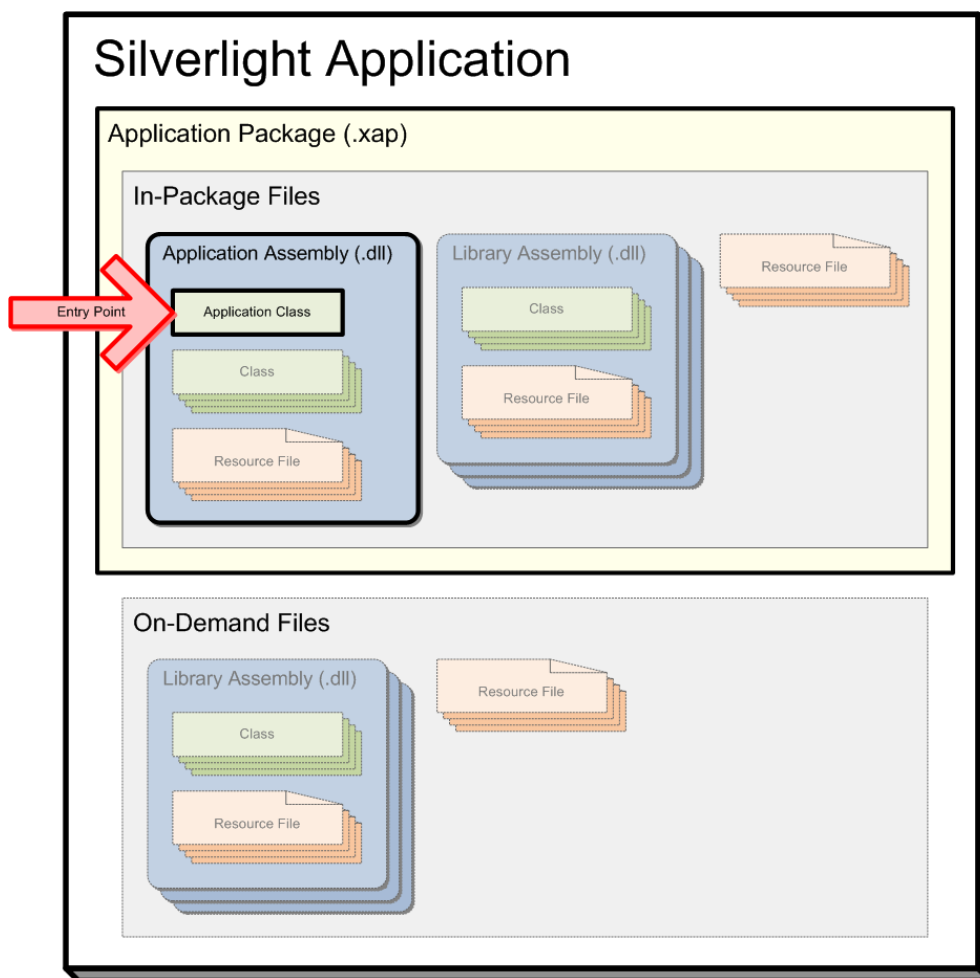
Térjünk át a Silverlight alkalmazás felépítésének tárgyalására! Itt hasonlóan az ASP.NET elgondoláshoz a megjelenés és a hozzá tartozó kód a *code-behind* modell alapján van szétválasztva. A felületet leíró kódok a *.xaml fájlokban találhatóak, a *.xaml.cs fájlok pedig a hozzájuk tartozó imperatív kódot tartalmazzák. Két ilyen fájlpárost kapunk alapból: egy **MainPage.xaml**-t és egy **App.xaml**-t. Előbbi az alkalmazásunk főoldala, ez jelenik meg alapból a böngészőben. Az **App.xaml** pedig leginkább az ASP.NET

projektekben használt **global.asax**-ra hasonlít abban az értelemben, hogy alkalmazásszintű események kezelését teszi lehetővé, illetve alkalmazásszintű erőforrások tárolására is alkalmas. Ez utóbbiakkal a stílusok használatánál, vagyis az 5. fejezetben foglalkozunk majd.

A **MainPage.xaml** és az összes többi oldal (navigációt nem használó alkalmazások esetén) valójában **UserControl** objektumok. Ez azért fontos, mert az újonnan így létrehozott oldalak vagy oldalrészletek könnyen beágyazhatók egy másik oldalba. Navigációs alkalmazás esetén a főoldalon egy **Frame** található, amelybe **Page** típusú objektumokat lehet betölteni. Az **App.xaml** vagyis az **App** osztály az **Application** *singleton* osztály egy leszármaztatott változata.

Érdekességgént megjegyezném, hogy van arra is lehetőség, hogy az eseményeket ne C# vagy Visual Basic nyelven kezeljük le, hanem JavaScript segítségével (Silverlight 1.0-ás hagyaték). Ilyenkor nincsen **partial class** segítségével megvalósított code-behind fájl, hanem helyette a hosztoló oldalon lehet azokat kezelni.

Maga a Silverlight alkalmazás, mint ahogyan azt már korábban is említettem, a hosztoló ASP.NET projekt **ClientBin** mappájába fordul le. Az alkalmazás a hozzá tartozó fájlokkal együtt egy zip fájlba becsomagolva kerül a mappába, **.xap** kiterjesztéssel. Erre azért van szükség, mert egy Silverlight alkalmazás weboldalba történő beágyazásakor csak egyetlen elemre szeretnénk hivatkozni, viszont egy Silverlight alkalmazás általában több elemből — egy vagy több dll-ből és gyakran erőforrásokból is, főleg képekből — áll. Ezt mutatja be az 1-12 ábra.



1-12 ábra: Egy Silverlightos XAP állomány felépítése és a futásidőben lekérhető egyéb erőforrások

Út a TestPage lekérésétől a MainPage Loaded esemény bekövetkezéséig

Ebben a részben annak járunk utána, hogy miként is indul be a gépezet — hogyan bögnek fel a motorok —, vagyis hogy milyen lépések sorozata vezet el ahhoz, hogy az alkalmazásunk elinduljon. Utunk során a **HelloSilverlight** alkalmazást szedjük apró darabokra, hogy végül összeálljon a teljes kép.

Kezdjük a legelején, kérjük le valamelyik **TestPage** oldalt! Ha megnézzük ennek az oldalnak a felületleíró kódját, és megkeressük benne a **<form>** elemet, akkor ott egy **<div>**-et és azon belül egy **<object>**-et találunk. Az **<object>** tag kódja az alábbi módon néz ki:

```
<object data="data:application/x-silverlight-2," type="application/x-silverlight-2">
  <param name="source" value="ClientBin/HelloSilverlight.xap"/>
  <param name="onError" value="onSilverlightError" />
  <param name="background" value="white" />
  <param name="minRuntimeVersion" value="4.0.50826.0" />
  <param name="autoUpgrade" value="true" />
  <a href="http://go.microsoft.com/fwlink/?LinkId=149156&v=4.0.50826.0"
    style="text-decoration:none">
    </a>
</object>
```

Ennek a deklarációnak a segítségével tudjuk azt megmondani, hogy egy Silverlight alkalmazást szeretnénk beágyazni az adott weboldalba. Erről az **<object>** elemről azt érdemes megjegyezni, hogy a **data** és a **type** attribútumok megadása kötelező. Ezenkívül, mint ahogyan az a fentebbi kódrészletből is látszik, a **param** tagok segítségével lehet az objektumot felparaméterezni. Az alapértelmezésben beállított tulajdonságok és jelentésük az alábbi:

- **source:** a lefordított Silverlight alkalmazás relatív helye
- **onError:** plug-in szintű hiba lekezelésekor lefuttatandó JavaScript kód
- **background:** alkalmazás háttérszíne
- **minRuntimeVersion:** az alkalmazás futtatásához szükséges legalacsonyabb Silverlight run-time verziószáma
- **autoUpgrade:** azt jelzi, hogy ha a kliensnél régebbi verziójú Silverlight van feltelepítve, mint ami az alkalmazás futtatásához szükséges, akkor megpróbálja-e a rendszer automatikusan frissíteni azt vagy sem.

Ennél jóval több plug-in szintű beállítás létezik, melyekkel a *Plug-in lehetőségek* című alfejezetben fogunk részletesebben foglalkozni.

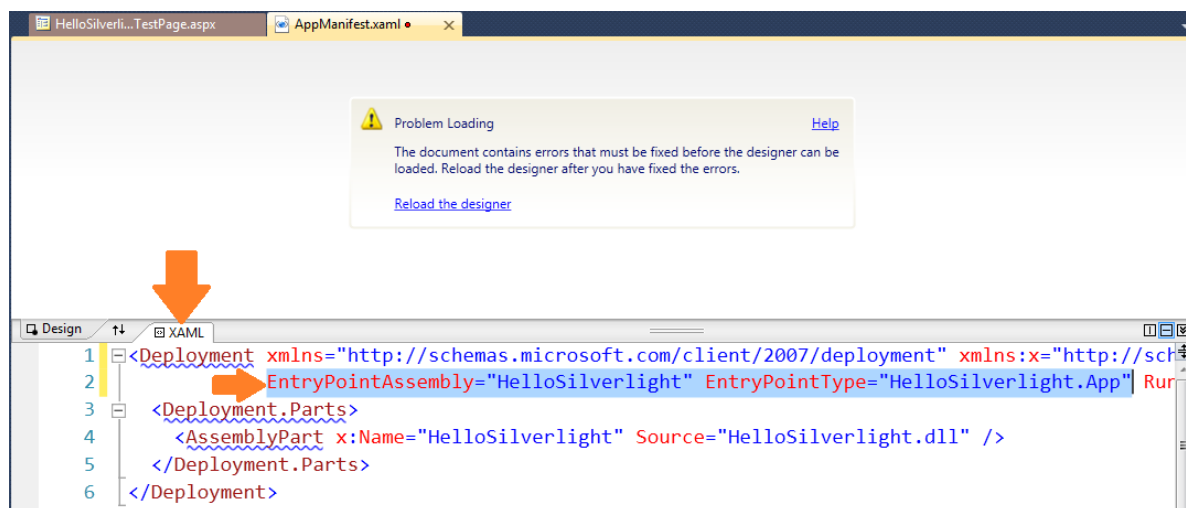
Az **<object>** nyitó és záró tagja között a **<param>** lista után található egy **<a>** tag. Ez a hivatkozás akkor jelenik meg, ha a felhasználó gépére nincsen feltelepítve a megfelelő Silverlight run-time, és alpból az 1-7 ábrán már látott képet jeleníti meg a rendszer. Ezt érdemes olyanra lecserélni, ami az adott alkalmazás kapcsán több információt hordoz. Például egy futás közbeni képernyőkép elhalványítva, melyre rá van írva a Silverlight run-time telepítésének szükségessége értelmesebben, magyarul megfogalmazva.

Most tehát ott tartunk, hogy lekértük az egyik **TestPage** fájlt, amelyben hivatkozunk a **.xap** állományra. Tegyük fel, hogy ez is letöltődött, ezért nézzük meg ennek a tartalmát! Nyissuk meg a **ClientBin** mappát Windows Intézőben (jobb klikk a mappán, majd Open Folder in Windows Explorer parancs), és készítsünk a fájlról úgy egy másolatot, hogy közben a kiterjesztését átírjuk **.zip**-re! A tömörített állomány tartalmaz egy **HelloSilverlight.dll**-t (az alkalmazás assembly) és egy **AppManifest.xaml** fájlt (az alkalmazás belépési pontját leíró fájl), mint ahogyan azt az 1-13 ábra is mutatja.

Name	Size	Packed	Type	Modified	CRC32
Folder					
AppManifest.xaml	373	208	Windows Markup File	2011.01.08. 22:51	708F1C56
HelloSilverlight.dll	8 192	3 451	Application Extension	2011.01.08. 22:51	CC4AA130

1-13 ábra: A XAP fájl tartalma

Az **Appmanifest** fájlt csomagoljuk ki valahová, majd töltjük be a Visual Studióba, így jobban olvasható lesz majd, mint ha csak szimplán egy jegyzetömbben nyitnánk meg. A megnyitáskor 3 hibaüzenetet fogunk kapni, és a XAML megjelenítő is hibát fog jelezni, de ezekkel ne törődjünk, egyszerűen csak kattintsunk duplán a XAML feliratra, mint ahogyan azt az 1-14 ábra is szemlélteti! (A hiba abból fakad, hogy a XAML egy sokkal általánosabb nyelv annál, mint hogy csak felület leírásra használjuk, viszont a Visual Studio felületleíróként próbálja meg értelmezni a XAML fájlban lévő kódot.)



1-14 ábra: AppManifest megnyitása Visual Studióban

A **Deployment** elem nyitó tagján belül található egy **EntryPointAssembly** és egy **EntryPointType** attribútum, mint ahogyan az 1-14 ábrán is látható. Ezek írják le az alkalmazás belépési pontját, vagyis hogy melyik szerelvényben melyik az az osztály, amely az **Application** típusból van származtatva. Ezen kívül a **Deployment.Parts** tag alatt fel vannak sorolva a .xap fájlban lévő könyvtárak és erőforrások is. Mivel a **HelloSilverlight** semmilyen extra dolgot nem használ, ezért ott most csak egyetlen bejegyzés található, az alkalmazás assembly.

Most, hogy már tudja a rendszer, hogy hol van az alkalmazás belépési pontja, már képes a futtatókörnyezet inicializálására. Nézzük meg, mi történik az **App** osztályon belül! Kezdjük a vizsgálatot a konstruktorral:

```
public App()
{
    this.Startup += this.Application_Startup;
    this.Exit += this.Application_Exit;
    this.UnhandledException += this.Application_UnhandledException;

    InitializeComponent();
}
```

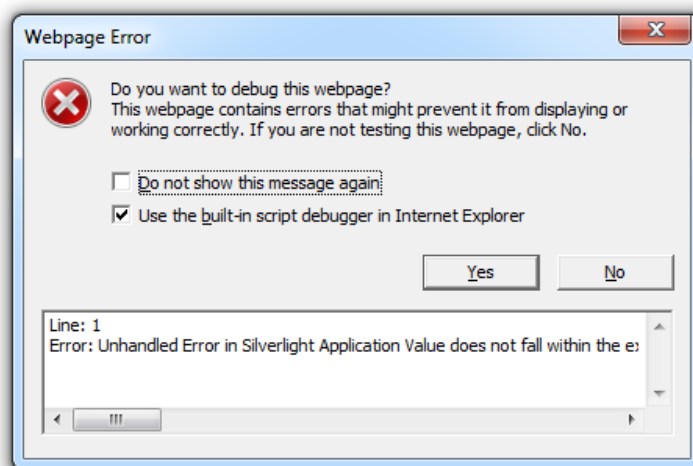
A kódrészletből az látszik, hogy az **App** osztály 3 eseményre iratkozik fel alaphól. A **Startup** esemény az alkalmazás indulásakor következik be, az **Exit** pedig akkor, amikor bezárjuk az alkalmazást, és végül az **UnhandledException**-höz rendelt eseménykezelő akkor kerül végrehajtásra, ha lekezeletlen hibát észlel

a rendszer. A konstruktor utolsó sora pedig egy olyan függvényt hív meg, amely a komponenseket inicializálja (erről majd kicsit később).

A **Startup**-hoz rendelt kód mindössze annyit csinál, hogy beállítja az alkalmazás kezdőoldalát, vagyis megadja, hogy az **App** osztály a **MainPage.xaml** fájlt jelenítse meg. Ezt az úgynevezett **RootVisual** tulajdonságon keresztül teszi, amelyet az alkalmazás életciklusa során csak egyszer lehet beállítani. Íme, az eseménykezelő kódja:

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    this.RootVisual = new MainPage();
}
```

Bár fel vagyunk iratkozva az **Exit** eseményre, de a hozzárendelt eseménykezelő függvény törzse mégis üres. Ellenben az **UnhandledException**-höz rendelt kód annál érdekesebb! Ugyanis egyrészt azt mondja, hogy kezeltnek tekinti a hibát (vagyis az alkalmazás képes innen továbbfutni), de emellett értesíti is a felhasználót arról, hogy valami hiba történt a rendszer futása közben. Ehhez egy új ablakot dob fel (amihez az HTML Bridge-t használja), és ebben tájékoztatja a felhasználót a történetekről, mint ahogyan azt az 1-15 ábra is mutatja. Természetesen éles környezetben célszerű ezt az alapértelmezett viselkedést lecserélni.



1-15 ábra: Kezeletlen kivétel jelzése

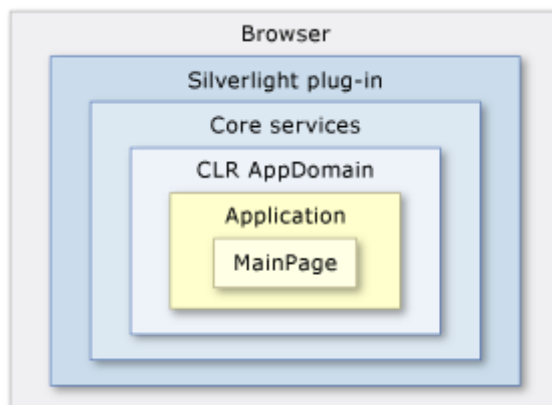
Mellesleg megjegyezném, hogy az alkalmazás szintű hibával ellentétben a Plug-in szintű hiba esetén az alkalmazás leáll, vagyis nincs lehetőség a hiba kezelése után a továbbfuttatásra.

Tehát most ott tartunk az alkalmazásunk életciklusában, hogy példányosítottuk a **MainPage** osztályt, és beállítottuk ezt a **RootVisual** tulajdonságnak. Tekintsük ezek után meg a **MainPage** betöltésének folyamatát! Ennek az osztálynak a konstruktora egyetlen metódushívást tartalmaz csak, mégpedig az **InitializeComponent**-et. A metódus forráskódját úgy tudjuk megtekinteni, hogy belekattintunk a függvény nevébe, majd megnyomjuk az F12-es gombot. Ilyenkor a Visual Studio átirányít bennünket a **MainPage.g.i.cs** fájlhoz, amelyet a rendszer automatikusan generál a háttérben. A függvény törzséből bennünket csak az alábbi pár sor érdekel:

```
System.Windows.Application.LoadComponent(this,
    new System.Uri("/HelloSilverlight;component/MainPage.xaml",
        System.UriKind.Relative));
this.LayoutRoot = ((System.Windows.Controls.Grid)(this.FindName("LayoutRoot")));
this.btn_hello = ((System.Windows.Controls.Button)(this.FindName("btn_hello")));
```


Ez a kódrészlet betölti a **MainPage.xaml** erőforrást a **HelloSilverlight** alkalmazás assemblyből, majd megkeresi benne a névvel ellátott (jelen esetben ezek a **LayoutRoot** és a **btn_hello**) vezérlőket, és eltárol róluk egy-egy referenciát. Erre azért van szükség, mert csak így tudjuk elérni a XAML-ben deklaratív módon létrehozott objektumokat C# kódból.

Ha mindez lefutott, akkor ezután a **MainPage** osztály **Loaded** eseménye is bekövetkezik, vagyis elindul az alkalmazásunk. Összefoglalásképpen az 1-16 ábra szemlélteti az alkalmazás betöltése során érintett egyes komponensek egymásba ágyazottságát.



1-16 ábra: Egy Silverlight alkalmazás betöltésében résztvevő komponensek egymásba ágyazottsága

Egy Silverlight alkalmazás fejlesztése esetén általában arra szokás törekedni, hogy minimalizáljuk a xap fájl méretét, vagyis az alkalmazás indítása a lehető leggyorsabb legyen. Erre sokféle technika áll rendelkezésünkre, melyek közül csak néhányat sorolnék fel a teljesség igénye nélkül:

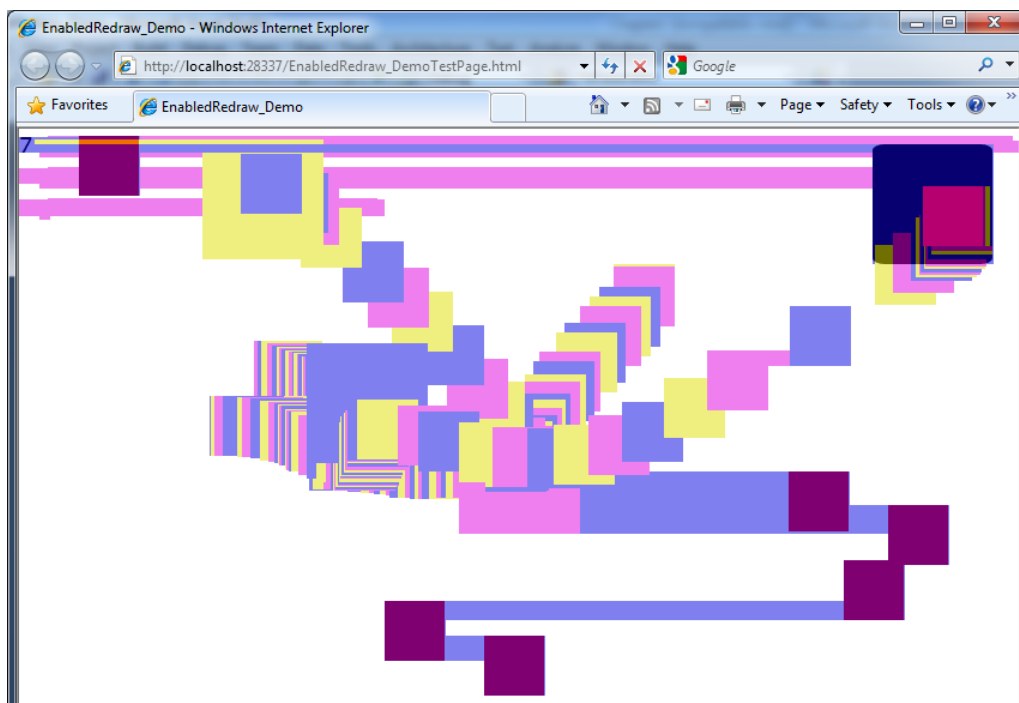
- *Application Library Caching* — a nem alkalmazás assembly, vagyis az osztálykönyvtárak gyorsítótárban tárolásának lehetősége.
- *Dynamic Assembly Loading* — futásidőben történő szerelvény le- és betöltése.
- *Managed Extensibility Framework* — moduláris alkalmazások fejlesztésének lehetősége.

Ezzel a kérdéskörrel részletesebben a 12. fejezetben foglalkozunk.

Plug-in lehetőségek

A plug-in testreszabhatóságából már kaptunk egy kis ízelítőt az előző alfejezetben, most ezt tárgyaljuk tovább. Csak néhány érdekesebb lehetőséget szeretnék bemutatni, a teljesség igénye nélkül:

- Az **enableFrameRateCounter** és **maxFrameRate** paraméterek — Ezek segítségével le tudjuk tesztelni, hogy például az animációk elérik-e a minimális 25 fps értéket, vagyis hogy folyamatosnak látszódnak-e. Bővebb információ itt található: <http://msdn.microsoft.com/en-us/library/system.windows.interop.settings.enableframeratecounter.aspx>.
- Az **enableRedrawRegions** paraméter — Ez hasonlóan az előzőhöz főleg animációknál használatos, ugyanis ezzel azt a szolgáltatást tudjuk bekapcsolni, amelyik képes megmutatni, hogy az egyes időpillanatokban a képernyő mely részeit kellett újrarajzolni a rendszernek. Az 1-17 ábrán egy játék látható futás közben, amelynél bekapcsoltuk ezt a szolgáltatást. A játék lényege, hogy az adott időközönként véletlenszerűen mozgó nagy négyzetbe (sötétkék) bele kell húzkodni a kisebb négyzeteket (lila). A különböző színek különböző iterációkat jelölnek.



1-17 ábra: Az *enabledRedrawRegions* paraméter engedélyezésének hatása

- Az *initParams* paraméter — Segítségével indulási paramétereket tudunk átadni a Silverlight kliensnek az őt hosztoló weboldalról. Ezeket a paraméterértékeket az **App** osztály **Startup** eseményében az **EventArgs** objektumon keresztül tudjuk lekérdezni kulcs-érték párok formájában. Bővebb információ itt található: [http://msdn.microsoft.com/en-us/library/cc189004\(v=vs.95\).aspx](http://msdn.microsoft.com/en-us/library/cc189004(v=vs.95).aspx).
- A *splashScreenSource* paraméter — Ha a xap fájlunk méretét valamilyen okból kifolyólag nem tudjuk 500 kilobyte alatt tartani, akkor célszerű ezt a szolgáltatást igénybe venni, ugyanis ezzel egyedi betöltési képernyőt adhatunk meg. Ehhez kapcsolódik két esemény is, a **PluginSourceDownloadProgressChanged** és a **PluginSourceDownloadCompleted**, melyek arról nyújtanak információt, hogy éppen hogyan áll a letöltés. Az 1-18 ábrán látható egy példa egyéni betöltési felületre. Bővebb információ itt érhető el erről a szolgáltatásról: [http://msdn.microsoft.com/en-us/library/cc903962\(v=vs.95\).aspx](http://msdn.microsoft.com/en-us/library/cc903962(v=vs.95).aspx).



1-18 ábra: A Silverlight Toolkit demó oldalának egyedi betöltési képernyője

A Plug-in lehetőségek teljes listája az alábbi weboldal bal oldali navigációs részén keresztül érhető el: [http://msdn.microsoft.com/en-us/library/cc838259\(v=VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc838259(v=VS.95).aspx).

XAML, a felületleíró nyelv

A pixelgrafika és a vektorgrafika összehasonlítása

Mielőtt belefognánk a XAML nyelv ismertetésébe, először meg kell értenünk, hogy egyáltalán miért is van szükségünk egy új nyelvre. Ehhez a pixeles (vagy raszteres) grafika és a vektorgrafika közötti különbségeket kell megvizsgálnunk.

A *pixelgrafika* úgy működik, hogy pontokat, pontosabban pixeleket helyezünk el a síkon úgy, hogy mindegyiknek megmondjuk a színét is, általában egy RGB színhármas (*Red, Green, Blue*) segítségével. Ezzel a megoldással minden egyes képpontra el kell tárolnunk egy színkódot.

A legtöbb képtömörítési algoritmus persze kb. 2-3-szoros arányban csökkenteni tudja a kép leírásához szükséges tárterületet. Az ún. veszteséges algoritmusok ennél sokkal jobb arányt (10-20-szoros méretcsökkenés) is el tudnak érni, de ez már ténylegesen rontja is a kép minőségét.

Mi a probléma ezzel a leírásmóddal? Alapjában véve semmi, de ha nagyítani vagy kicsinyíteni kell a képet, akkor már bajban vagyunk. Ugyanis a nagyítás mértékétől függően új képpontok jönnek létre, amelyek az eredeti képen még nem szerepeltek, emiatt a színük ismeretlen. Tehát a rendszernek kell a környezetből kikövetkeztetnie az új pixelek színeit. Ennek következtében a képek homályosak, elmosódottak és „töredezettek” lesznek, mint ahogyan azt az 1-19 ábra is mutatja.



1-19 ábra: Pixelgrafika - 14-es betűméretű szöveg 15-szörös nagyítással

A *vektorgrafika* vagy geometriai modellezés ezzel szemben nem pixelekkel dolgozik, hanem geometriai primitívekkel (például pontokkal, egyenesekkel, görbékkel). Ez azzal az előnnyel jár, hogy matematikai transzformációk segítségével könnyen nagyíthatóak, kicsinyíthetőek, sőt nyújthatóak, dönthetőek is az alakzatok. Az 1-20 ábra az 1-19 ábra vektorgrafikus megfelelőjét mutatja.



1-20 ábra: Vektorgrafika – 14-es betűméretű szöveg 20-szoros nagyítással

A vektorgrafikus alakzatok könnyen egymásba ágyazhatóak, és minden egyes elemről egyértelműen megmondható, hogy ki a szülő vagy konténer alakzata. Tehát minden elemnek legfeljebb **egy** közvetlen szülője lehet. Ez azt eredményezi, hogy könnyen hierarchiába szervezhetőek, vagyis XML alapú reprezentációval jól leírhatók. Az XML mellett szól még az is, hogy az alakzatokat le lehet írni deklaratív módon is (objektum + tulajdonsághalmaz → tag + attribútumok). Ez az elgondolás vezetett el az XAML nyelv megalkotásához.

A XAML nyelv alapjai

A XAML az *Extensible Application Markup Language* (~kiterjeszthető alkalmazás leíró nyelv) rövidítése. A Silverlightban, illetve a WPF-ben elsősorban a felületleírásra használják, de mint ahogy azt láthattuk, az

AppManifest is ebben a formátumban tárol el információkat a Silverlight alkalmazásról, illetve a *Workflow Foundation* is ezt a nyelvet használja a folyamatok gráfjának leírására.

Ahogy a neve is utal rá, az XAML egy XML (*eXtensible Markup Language*) alapú nyelv. Ez azt jelenti, hogy az XML-re vonatkozó szabályok itt is érvényesek. Néhány fontos dolog, amit az XML-ről tudni kell:

- Deklaratív nyelv, vagyis azt írjuk le benne, hogy **mit** szeretnénk, és nem azt, hogy **hogyan** (a hogyan kérdéskörrel az imperatív nyelvek foglalkoznak).
- Az elemeket (tagokat) kisebb („<”) és nagyobb („>”) jelek közé írt azonosítók segítségével tudjuk leírni. Ezekből van egy nyitó és egy záró változat, pl.: `<demo>...</demo>`.
- A nyitó és záró elemek között el lehet helyezni tartalmat, illetve a nyitó tageknek lehetnek attribútumai (kulcs-érték párok), például: `<demo id="1">XML DEMO</demo>`.
- Az elemek egymásba ágyazhatóak, vagyis hierarchiába szervezhetőek. Például: `<a><b></b><b><c></c></b></a>`.
- Az XML fájlhoz (dokumentumhoz) tartozhat egy definíciós vagy séma fájl, amely az egyes elemek szerkezetét, illetve azok hierarchiáját írja le. A séma alapján lehet *érvényesíteni* őket.
- Egy XML fájlnak meg kell felelnie néhány követelménynek, amelyeket együttesen *jól formázottsági kritériumoknak* neveznek. A teljesség igénye nélkül, íme, néhány: a dokumentumnak csak egyetlen gyökéreleme lehet, minden megkezdett taget le kell zárni.
- A nyelv kis- és NAGYbetű érzékeny.
- Bővebb információ az XML-ről itt található: <http://hu.wikipedia.org/wiki/XML>.

A XAML nyelv szintaxisa

Ebben a részben az objektumok leírási módjaival, pontosabban fogalmazva a leírási lehetőségekkel foglalkozunk. Íme, egy rövid lista ezek bemutatására — a teljesség igénye nélkül:

- **Objektum szintaxis:** nyitó és záró elemek segítségével leírt objektum deklaráció, például:

```
<Button> </Button>.
```

Ez a kódrészlet rövidíthető, ha nem akarunk a két tag közé semmilyen tartalmat elhelyezni, akkor így is leírhatjuk:

```
<Button />.
```

- **Attribútum szintaxis:** az objektum egy adott tulajdonságának beállítása, például:

```
<Button Content="Hello"></Button>
```

- **Literális érték leírása:** tartalom megadására használható forma, például:

```
<Button>Hello</Button>
```

(Mellesleg ez a kód teljesen ekvivalens az előző példakóddal).

- **Tulajdonság szintaxis:** objektum tulajdonságainak beállítására szolgál, főleg összetett értékek esetén használatos, például:

```
<Button>
  <Button.Content>
    Hello <Run FontWeight="Bold">Silverlight</Run>
  </Button.Content>
</Button>
```

(Megjegyzés: a **Run** objektummal lehet szöveget formázni.)

- *Konténer szintaxis:* listaelemek felsorolására használható forma, például:

```
<Button.Fill>
  <LinearGradientBrush >
    <LinearGradientBrush.GradientStops >
      <GradientStop Offset="0.0" Color="Red" />
      <GradientStop Offset="1.0" Color="Blue" />
    </LinearGradientBrush.GradientStops >
  </LinearGradientBrush>
</Button.Fill>
```

Megjegyzés: ez a kód a gombot egy átlós piros-kék színátmenettel tölti ki. Ezzel a technikával részletesebben a 4. fejezetben foglalkozunk.

Lehetőség van arra is, hogy objektumokat névterekbe ágyazzuk. Egy másik (vagyis nem az alapértelmezett) névtérben lévő objektum használatához először egy névtér regisztrációt kell elhelyeznünk a XAML dokumentum elején, majd ezen keresztül kell az adott objektumra hivatkoznunk. Például:

```
<UserControl xmlns:this="myNameSpaceLocation">
  ...
  <this:MyControl />
  ...
</UserControl>
```

Megjegyzés: az `xmlns:` utáni szócska tetszőlegesen választható azonosító

További XAML lehetőségek

Itt néhány olyan fogalmat, technikát sorolunk fel, amelyeket érdemes még a XAML nyelvről tudni:

- *Markup Extensions:* a leíró nyelv kiegészítései, amelyek speciális feldolgozást igényelnek az XAML parser (értelmező) részéről. Ilyenek például a **StaticResource** (lásd 4., 5. fejezet), **Binding** (lásd 6. fejezet), **TemplateBinding** (lásd 5. és 7. fejezet).
- *Keresés az XAML hierarchiában:* ha egy konkrét gyerek elemére vagyunk kíváncsiak, akkor a **FindName** metódus segítségével kereshetjük meg az adott elemet, ha viszont egy összetett objektum gyerekeit szeretnénk elérni, akkor a **GetTemplateChild** metódusra lesz szükségünk.
- *Dinamikus betöltés:* .NET kódból a **XAMLReader** osztály **Load** metódusával, JavaScript kódból pedig a **CreateFromXaml()** függvénnyel tölthetünk be egy XAML nyelven leírt kódrészletet.
- *Inline XAML:* a hosztoldalon közvetlenül is elhelyezhető XAML kód, ilyenkor a **<script type="text/xaml">...</script>** elemek közé kell elhelyezni a tartalmat, amelyet JavaScript kódból vagy valamilyen dinamikus nyelvből el lehet érni, esetleg .NET kódból a HTML Bridge segítségével.
- Az XAML nyelvről bővebb információ az alábbi weblapon található:
<http://msdn.microsoft.com/en-us/library/ms788723.aspx>.

Összefoglalás

Ebben a fejezetben megismerkedtünk a Silverlight alapjaival, illetve felépítésével, szolgáltatásaival és működésének módjával. Eközben megvizsgáltuk azt is, hogy milyen eszközök szükségesek a Silverlight alkalmazások fejlesztéséhez, illetve milyen projektípusokat tudunk használni a Visual Studióban. A fejezetet a Silverlight megjelenítő rendszerének leíró nyelvével (az XAML-lel) zártuk.

2. Layout Management

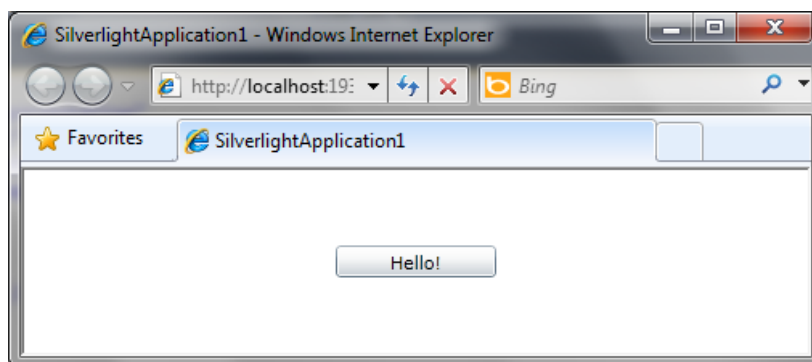
Amikor layout-managementről beszélünk, arra a folyamatra gondolunk, amely az alkalmazásunk felhasználói felületén megjelenített vezérlőelemek elrendezéséért felelős. A Silverlight rendkívül rugalmas környezetet biztosít számunkra, amely egyaránt lehetővé teszi statikus vagy akár a böngésző méretéhez alkalmazkodó felhasználói felület készítését.

Pozicionálás

A Silverlight mind az abszolút – koordináta alapú –, mind a dinamikus (ún. *self-sizing*) pozicionálást támogatja. A kettő közül az utóbbi rugalmasabb (ez az „ajánlott”), emellett a megjelenítést leíró kód is sokkal áttekinthetőbb.

Minden Silverlight program egy (és csakis egy) konténer elemre építkezik, amely a vezérlőelemeket tárolja. Alapértelmezés szerint minden vezérlőelem az őt tároló konténer viselkedése alapján kap helyet, pl. a **Grid** az aktuális középpontra, míg a **Canvas** a bal felső sarokba pozicionál.

A 2-1 ábrán egy Grid vezérlőelem látható egyetlen gombbal.



2-1 ábra: Grid vezérlő egy gombbal

Nézzük meg a vezérlőelemet leíró XAML kódot:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">
    <Grid x:Name="LayoutRoot">
        <Button Width="100" Height="20" Content="Hello!" />
    </Grid>
</UserControl>
```

A vezérlők – amennyiben nem állítottuk be külön a méreteiket – kitöltik a rendelkezésükre álló teret, ez alól egyetlen kivétel van, mégpedig az, ha a konténer elem a **Canvas**.

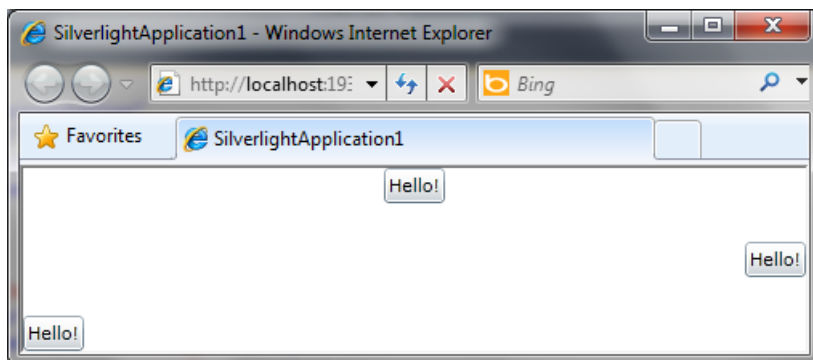
Természetesen lehetőségünk van a pozíciók finomhangolására is, de ez az egyes konténer-vezérlőknél más-más módon történik, őket majd a megfelelő fejezetben vizsgáljuk meg közelebbről. Addig is rendelkezésünkre áll egy egyszerűbb módszer, mégpedig a **VerticalAlignment** és **HorizontalAlignment** tulajdonságpáros, amelyeket minden vezérlő birtokol. Tulajdonképpen eddig is

használtuk őket (még ha ez a kódban nem is látszik), mivel alapértelmezés szerint – pontosabban, ha nem adtuk meg explicit módon a szélességet illetve magasságot – mindkét tulajdonság a **Stretch** értékkel rendelkezik, ami a fent már említett jelenséget eredményezi, vagyis a vezérlő az egész teret elfoglalja.

Nézzünk meg egy egyszerű példát, legyen az XAML a következő:

```
<Grid x:Name="LayoutRoot">
  <Button Content="Hello!" VerticalAlignment="Top" HorizontalAlignment="Center" />
  <Button Content="Hello!" VerticalAlignment="Center" HorizontalAlignment="Right" />
  <Button Content="Hello!" VerticalAlignment="Bottom" HorizontalAlignment="Left" />
</Grid>
```

Az eredményt a 2-2 ábra mutatja.



2-2 ábra: A *VerticalAlignment* és *HorizontalAlignment* hatása

Vegyük észre, hogy bár nem adtunk meg a gomboknak külön méretet, a **Stretch** felülírásával a vezérlő már nem akar agresszívan terjeszkedni, beéri kevesebb helyel is!

Layout életciklus

A vezérlők megjelenítése a felhasználói felületen lényegében két lépésre bontható le, de ez ne tévesszen meg senkit, hiszen megfelelő körütekintés hiányában teljesítményproblémákba is ütközhetünk!

Measure

Az első lépés a sorban az ún. „*Measure-pass*” (méretezési fázis), amelynek során a szülő objektum (valamelyik konténer elem) átadja minden gyermek elemének (valamelyik vezérlő) a számára kijelölt terület méretét. Ezután a vezérlő – figyelembe véve a saját elképzeléseit, vagyis a megadott szélesség, magasság és egyéb paramétereket – kiszámolja, hogy mekkora területen kíván elhelyezkedni, és ezt az adatot eltárolja magának (egyúttal lehetősége van arra is, hogy a megadottnál nagyobb területet igényeljen).

Arrange

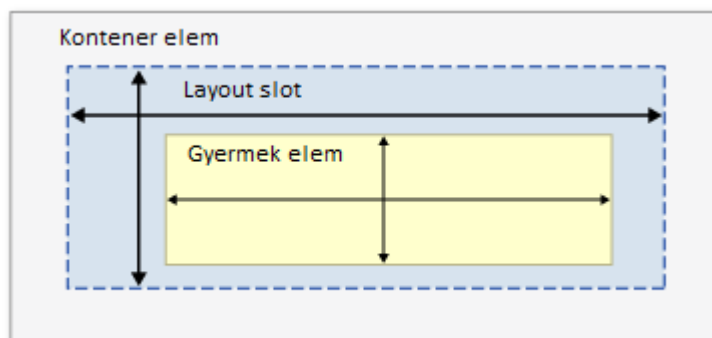
A Measure-pass után következik az „*Arrange-pass*” (elrendezési fázis), ekkor a rendszer minden egyes gyermekelemnek átad egy **Rect** típusú objektumot (amely x és y koordinátákat, illetve a szélesség és magasság adatokat tartalmazza) kijelölve ezzel a pozíciót és a vezérlő méretét is. Ezeket az adatokat a Measure-pass során megszerzett információkból számolja a rendszer.

Bounding-Box

Mivel a vezérlők többsége nem szabályos téglalap alakú (eleve egy gomb is inkább összenyomott ellipszist formál), ezért nehezen tudjuk őket pozicionálni, mivel nincs egyértelmű viszonyítási pont a koordinátákhoz. Általában a „bal felső sarok” szolgál erre a célra, de egy „díszos” lekerekített sarkokkal szerelt vezérlőnél ez a fogalom értelmét veszti. Erre a problémára jelent megoldást a *bounding-box* használata, vagyis ahelyett, hogy közvetlenül a vezérlőknek adjuk meg a pozíciójuk koordinátáit, inkább

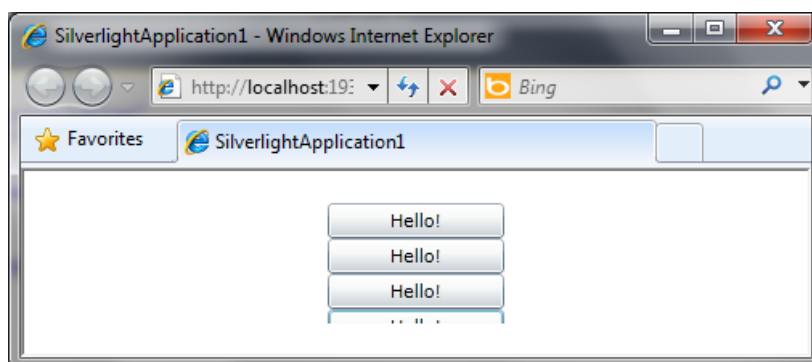
egy körékük írható téglalapot használunk erre a célra, miáltal minden vezérlőhöz használható egységes módszerhez jutunk.

A téglalap által határolt területet az adott vezérlő *layout-slot*-jának nevezzük, ezt a szerkezetet a 2-3 ábra mutatja be.



2-3 ábra: A Layout-slot

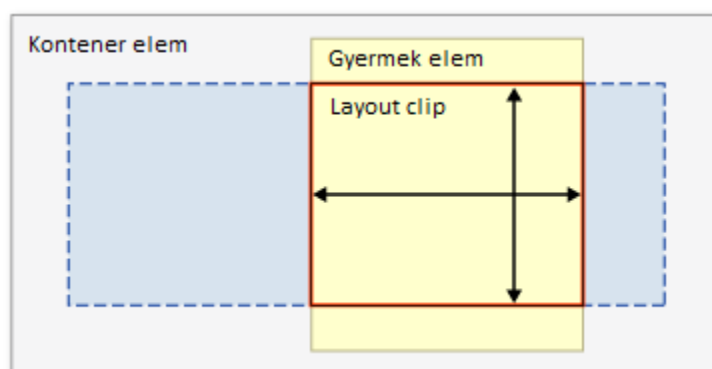
A layout-slot mérete a *measure/arrange* lépésekben alakul ki, és nem feltétlenül biztosít elég helyet. Ekkor a vezérlő egész egyszerűen kilóg majd a layout-slotból, a túlnyúló régiókat nem fogjuk látni, amint azt a 2-4 ábra is mutatja.



2-4 ábra: A vezérlő kilóg a Layout-slotból

A képen látható alkalmazásban a **StackPanel** konténert használtuk, szándékosan kevés helyet hagyva, így megfigyelhetők, hogy az alsó gombnak kiosztott layout-slot kevésnek bizonyult.

A layout-slotból kilógó elemek látható részét *layout-clip*-nek nevezzük, ennek „működési elvét” mutatja be a 2-5 ábra.



2-5 ábra: A Layout-clip

A „clipesedés” két módon jöhet létre: az egyik a fent bemutatott eset, amikor eleve kevés a hely. A másik egy utólagos módosítás, jellemzően valamilyen animáció hatására alakul ki, például elforgatunk egy vezérlőt.

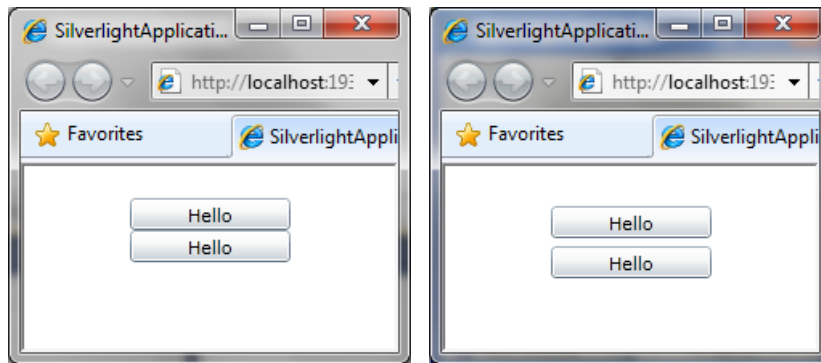
A layout-slot és layout-clip információk elérhetőek a **LayoutInformation** osztály statikus **GetLayoutSlot** és **GetLayoutClip** metódusaival.

Margin és Padding

Minden vezérlőn állíthatóak a **Margin** és **Padding** tulajdonságok, amelyek a bounding-boxhoz köthetőek. Előbbi a vezérlők közti távolságot állítja be, nézzünk egy példát:

```
<StackPanel Width="110" Height="75" x:Name="LayoutRoot">
  <Button Width="100" Height="20" Margin="5" Content="Hello"></Button>
  <Button Width="100" Height="20" Content="Hello"></Button>
</StackPanel>
```

Ebben az esetben az uniform értékadást használtuk, vagyis a vezérlő négy oldalán ugyanakkora margót hagyunk. A 2-6 ábrán jól látható a változás (a bal oldali képen még nem használtuk a **Margin**-t).



2-6 ábra: A Margin és Padding tulajdonságok hatása

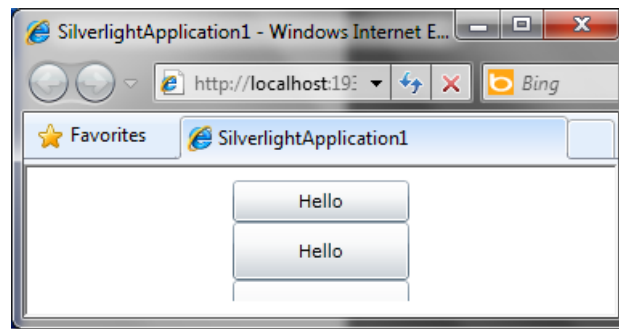
A **Margin** nem a bounding-boxot terjeszti ki, hanem azon kívül hagy extra helyet.

Háromféleképpen definiálhatjuk az értékeket: az első a fenti példában használt uniform módszer, amely egyetlen – **double** típusú – számot vár, ezt alkalmazza a vezérlő minden oldalára a rendszer (valójában a **Margin** tulajdonság egy **Thickness** struktúrában tárolja az adatokat). Ezenkívül az értéket megadhatjuk páronként (főnt-lent és jobb-bal), illetve egyenként kifejtve is (ekkor kilenc óránál kezdünk, és az óramutató szerint haladunk: bal-főnt-jobb-lent):

```
<Button Margin="2" Content="Hello" />
<Button Margin="2, 4" Content="Hello" />
<Button Margin="2, 4, 2, 4" Content="Hello" />
```

A **Padding** épp az ellenkezőjét teszi, a vezérlő és a bounding-box „keret” távolságát szabályozza. Hatására az adott vezérlő „megnö”, és ha nincs számára elég hely, akkor „kinyomja” a konténerben lévő többi vezérlőt, vagyis a helyfoglalásban prioritást élvez. Az alábbi kód hatását a 2-7 ábra mutatja meg.

```
<Button Padding="5" Content="Hello" />
<Button Padding="5, 10" Content="Hello" />
<Button Padding="5, 10, 5, 10" Content="Hello" />
```



2-7 ábra: a Padding hatása

Layout metrika

Ha a monitor fizikai pixelei alapján pozicionálnánk, előbb vagy utóbb bajba kerülnénk, mivel az alkalmazásunk – felbontástól függően – túl nagy vagy túl kicsi lenne. Erre a problémára már régóta létezik a megoldás, mégpedig a DIP (*Device Independent Pixel*) formájában.

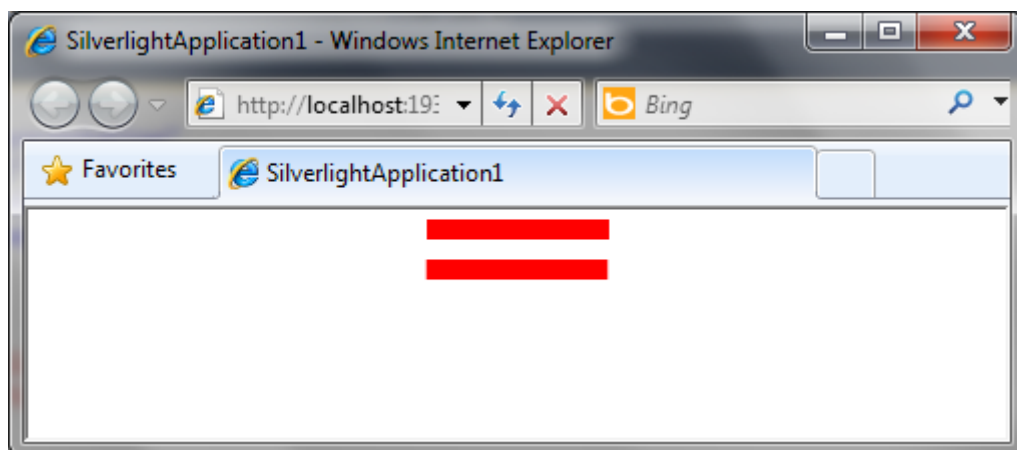
A jelenleg használt metrika alapját a hüvelyk (inch) képezi, amely 2,54 cm hosszúságú. Egy logikai hüvelyk mindig 96 DIP, ezáltal egy logikai pixel valós méretét megkapjuk, ha az aktuális DPI (*Dot Per Inch* – az egy hüvelyken lévő pixelek száma) számot elosztjuk 96-tal. A DPI növelésével a DIP és a fizikai pixelek aránya is változik. Az alapbeállítás a legtöbb számítógépen 96 DPI, ekkor egy DIP egy fizikai pixelnek felel meg. Ha átváltunk pl. 144 DPI-re, akkor egy DIP már másfél fizikai pixelt foglal ($144 / 96 = 1,5$), tehát egy 100x100-as négyzet 150x150 méretűre hízik – fizikailag; a logikai – általunk megadott mérete – változatlan marad.

Layout rounding

A vezérlők elrendezésekor előfordul, hogy a szélesség/magasság tulajdonságokhoz vagy a koordináták értékeihez nem egész szám kerül, ekkor a vezérlőt a rendszer úgy rajzolja meg, hogy bizonyos részei nem egész pixeleket foglalnak el (subpixel rendering). Készítsünk egy egyszerű felületet:

```
<StackPanel x:Name="LayoutRoot">
    <Rectangle UseLayoutRounding="True" Margin="5" Fill="Red" Width="90.5" Height="10" />
    <Rectangle UseLayoutRounding="False" Margin="5" Fill="Red" Width="90.5" Height="10" />
</StackPanel>
```

Egyelőre ne törődjünk azzal, hogy mit miért írtunk, inkább lássuk az eredményt, amit a 2-8 ábra mutat:



2-8 ábra: Subpixel rendering madártávlattól

Első ránézésre hasonló a két „vonal”, de ha nagyítunk egy kicsit a képen, akkor feltárul az igazság:



Az alsó vonal két szélén világosabb régiók jelentek meg, ezt az *anti-aliasing* effektnek „köszönhetjük”, amely a nagy felbontásban „kilógó” régiókat (jelen esetben a nem egész pixeleket) semlegesíti, hogy kis felbontásban szebb legyen az eredmény (az *anti-aliasing* témaköre ennél persze jóval bonyolultabb). A probléma az, hogy ezt mi nem minden esetben szeretnénk, mert az eredmény nem kell feltétlenül szép legyen, valamint ez a vezérlők méretét is növelheti, ezáltal széteshet a felület.

Térjünk vissza a fenti XAML kódhoz és vizsgáljuk meg jobban! A **UseLayoutRounding** tulajdonság gondoskodik arról, hogy kerek értékeket használjunk, vagyis a vezérlő határait a legközelebbi pixelig kerekíti.

Teljesítmény

A Measure és Arrange lépések meglepően sarkalatos pontjai az alkalmazásunknak. Nézzünk egy egyszerű példát: egyetlen **Button** vezérlőt jelenítünk meg, a kérdés a következő: hány Measure/Arrange fázisra számíthatunk?

Látszólag egyértelmű a válasz, a valóság azonban kegyetlenebb. A Silverlight összes vezérlője sablonok szerint épül fel, és ezek alapja szintén egy-egy konténer elem. Vagyis minden elem kisebb részekből áll, amelyekre meg kell hívni egy-egy Measure/Arrange metódust egészen a legbelső – tovább már nem bontható – körig.

Visszatérve az eredeti kérdésre, egyetlen, egyszerű **Button** vezérlő esetében nagyjából egy tucat metódushívással kell számolnunk. Ez persze még egy viszonylag összetett űrlap futtatásánál sem jelent valós problémát, de vannak helyzetek, amikor oda kell figyelnünk.

Lássuk, hogy hogyan vehetjük elejét a gondoknak!

Virtualizáció

Tipikusan a listaszerű vezérlők sajátossága, hogy nagy elemszámnál jelentős lassulással kell számolni. Hogy ez miért történik, arra legegyszerűbben egy példán keresztül jöhetünk rá: képzeljünk el egy **ListBox** elemet ezer elemmel, ez egyáltalán nem irreális mennyiség. Nyilvánvaló, hogy ezt az ezer elemet nem tudjuk egyszerre megmutatni, vagyis a listát „mozgatni” kell. A probléma itt keletkezik, és rögtön két oldalról is támad: egyrészt a Silverlight minden vezérlőeleme – beleértve a listaelemeket is – sablonokból épül fel, amelyek maguk is kisebb vezérlőkből állnak, és így tovább. Természetesen ekkor az összes részegységre meg kell hívni a Measure/Arrange párost. A másik gondunk pedig az, hogy az ezer listaelemnek memóriát is kell foglalni, márpedig ezer elemet a memóriában tartani, amikor egyszerre — mondjuk — csak ötven látszik, nem túl okos dolog.

Szerencsére a Silverlight rendelkezik megoldással, rögtön kétféle stratégia közül is választhatunk. Az alapértelmezés szerint minden listaelemnek csak akkor foglalunk memóriát, ha az éppen látszik. Ezzel a memóriaproblémát ugyan megoldottuk, cserében a processzor életét nehezítjük, hiszen ekkor a memóriaallokálás mellett a Measure/Arrange párossal is számolni kell. A másik módszer, az ún. *control recycling* ehelyett azt mondja, hogy úgyis készítettünk annyi listaelemet, amennyire szükségünk van, miért dobánk ki őket? Tehát egész egyszerűen a vezérlők a helyükön maradnak, és csak az adatokat cserélgetjük.

A Silverlight tartalmaz beépített „virtualizált” konténert, a **VirtualizedStackPanel**-t, ezt jó néhány beépített vezérlő, pl. a **ListBox** is használja. A következő kódban ezen a vezérlőn próbáljuk ki a két stratégiát:

```
<ListBox VirtualizingStackPanel.VirtualizationMode="Standard" />
<ListBox VirtualizingStackPanel.VirtualizationMode="Recycling" />
```

Transzformációk

Minden olyan esetben, amikor egy vezérlő felülethez kötődő tulajdonságát (pl. szélesség, magasság) megváltoztatjuk, automatikusan meghívódik a `Measure/Arrange` páros is. Ezt elkerülhetjük azzal, ha közvetlen módosítás helyett transzformációkat/animációkat használunk.

Layout vezérlők

Minden konténer-vezérlő az absztrakt **Panel** osztályból származik. A vizuális megjelenést leszámítva, a „családfa” alapján ezek ténylegesen vezérlők, azaz minden olyan tulajdonsággal és eseménnyel rendelkeznek, mint a „hagyományos” társaik.

Grid

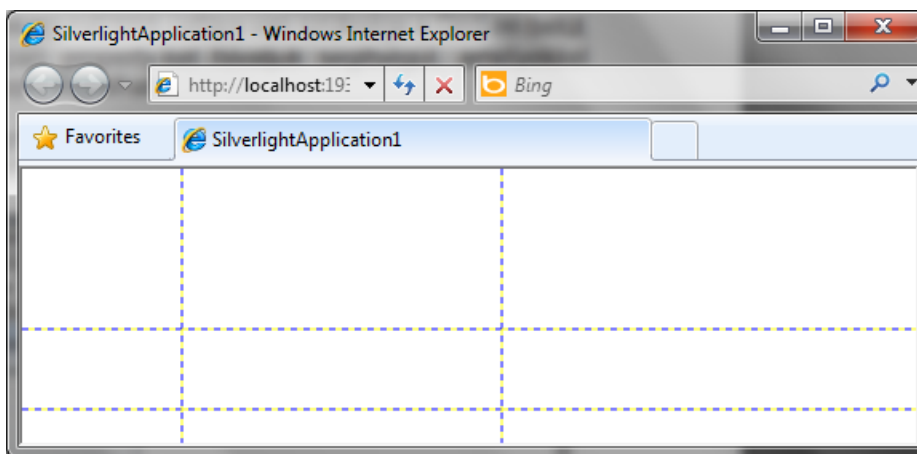
A **Grid** a legsokoldalúbb és ezáltal leggyakrabban használt konténer. Ez a vezérlő lehetővé teszi tetszőleges számú és méretű oszlop illetve sor definiálását. Alapértelmezés szerint egy sor és egy oszlop létezik, amennyiben többet szeretnénk, megfelelő számú **RowDefinition** és **ColumnDefinition** objektumot kell készítenünk. Nézzünk egy példát:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="400">

  <Grid x:Name="LayoutRoot" ShowGridLines="True">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="100" />
      <ColumnDefinition Width="200" />
      <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="100" />
      <RowDefinition Height="50" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
  </Grid>
</UserControl>
```

A **ShowGridLines** tulajdonság állításával láthatóvá tehetjük a **Grid** oszlopait és sorait határoló vonalakat. Látható, hogy három-három egységre bontottuk a **Grid**-et, és kettő kivételével explicit módon megadtuk a méretet is (amennyiben ez utóbbit nem tesszük meg, akkor a definíciók között egyenlően oszlik el a felület).

A fenti kód eredményét a 2-9 ábra mutatja meg.



2-9 ábra: A Grid szerkezete

Érdeemes megfigyelnünk, hogy az alsó sor és a jobb szélső oszlop magasságát illetve szélességét „csillaggal” jelöltük az ún. *star-sizing* módszerrel. Ennek lényege, hogy a megmaradó helyet a csillagos definíciók között súlyozottan osztja el a rendszer. Mit értünk ez alatt? Opcionálisan megadhatunk egy egész számot, amely a sorra/oszlopra eső méret arányát jelöli, például:

```
<Grid.RowDefinitions>
  <RowDefinition Height="100" />
  <RowDefinition Height="1*" />
  <RowDefinition Height="2*" />
</Grid.RowDefinitions>
```

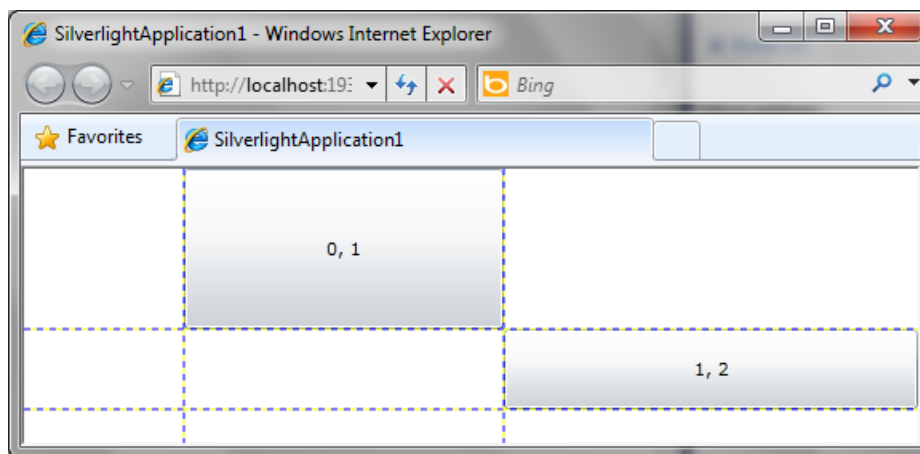
Ebben az esetben az első sor által elfoglalt helyet nem számítva a másik kettő egy- illetve kétharmadnyi területen osztozódik. Legyen pl. a **Grid** magassága 400, ekkor az első (csillaggal jelölt) sor 100, a második 200 magasságú területtel gazdálkodhat.

A csillag-jelölés mellett „Auto”-ra is állíthatjuk az értéket, ekkor annyi helyet kap az adott sor/oszlop, amennyi a benne lévő vezérlőknek szükséges.

Oszlopokat és sorokat már tudunk készíteni, a következő lépés az, hogy vezérlőket helyezünk el bennük. Ehhez a művelethez egy speciális tulajdonságot — az *attached-property*-t — használunk fel, amely — ebben az esetben — lehetővé teszi, hogy egy vezérlőelem az őt tároló konténer egy tulajdonságát használja:

```
<Grid x:Name="LayoutRoot" ShowGridLines="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="200" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="100" />
    <RowDefinition Height="50" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Button Content="0, 1" Grid.Row="0" Grid.Column="1" />
  <Button Content="1, 2" Grid.Row="1" Grid.Column="2" />
</Grid>
```

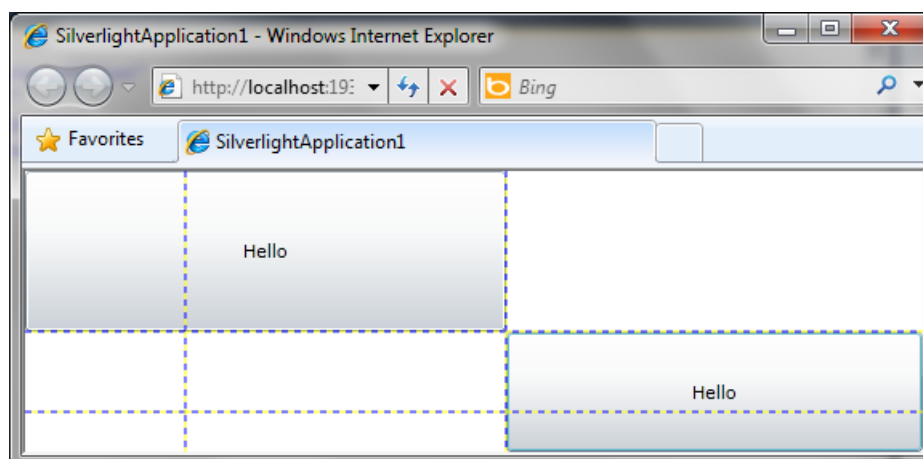
A vezérlőknél helyben kell kifejtenuünk mind az oszlop, mind a sor számát (természetesen nullától kezdődik az index), amennyiben az egyik hiányzik, akkor automatikusan nulla értéket kap, vagyis a legelső sorba/oszlopba kerül. A fenti definíció hatását a 2-10 ábra mutatja be.



2-10 ábra: A Grid celláinak számozása

Előfordul, hogy egy-egy vezérlőnek egynél több cellára van szüksége a **Grid**-en belül, ekkor a **ColumnSpan** és **RowSpan** tulajdonságokat hívhatjuk segítségül, amelyekkel beállíthatjuk, hogy az hány sort/oszlopot foglalhat (2-12 ábra):

```
<Button Content="Hello" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2" />
<Button Content="Hello" Grid.Row="1" Grid.Column="2" Grid.RowSpan="2" />
```



2-11 ábra: A RowSpan és ColumnSpan hatása

Canvas

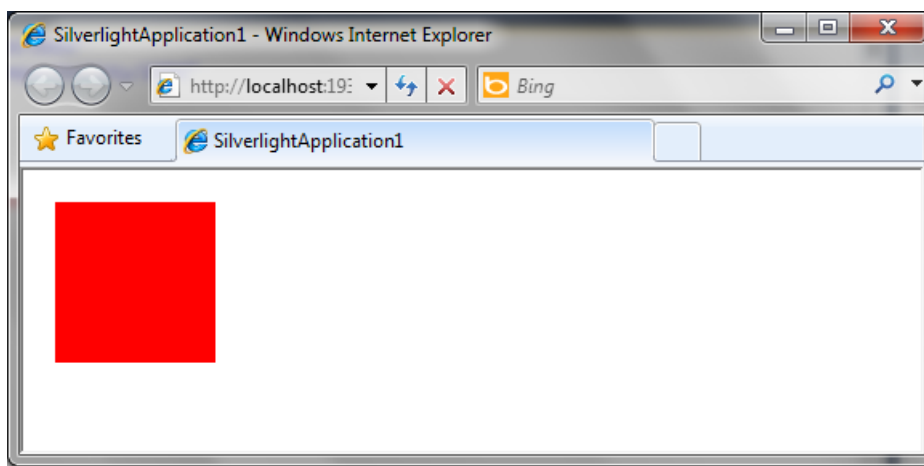
A **Canvas** lehetővé teszi, hogy explicit módon koordináták segítségével pozicionáljuk a vezérlőinket. Fontos tudni, hogy nem veszi figyelembe a böngésző méretét, tehát általában nem jó választás legfelsőbb szintű konténernek, jellemzően egy másik **Panel** leszármazottba ágyazzuk be.

Az x/y koordinátákat a **Canvas.Left** illetve a **Canvas.Top** attached propertykkel állítjuk be, amelyekkel a **Canvas** bal oldalának és felső részének a bounding boxtól való távolságát szabályozzuk. Amennyiben ezeket az értékeket nem adjuk meg, akkor automatikusan a bal felső sarokba (0;0) pozicionál. Az alábbi definíció hatását a 2-12 ábra mutatja meg:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Canvas x:Name="LayoutRoot">
        <Rectangle Width="100" Height="100" Fill="Red"
            Canvas.Left="20"
            Canvas.Top="20" />
    </Canvas>

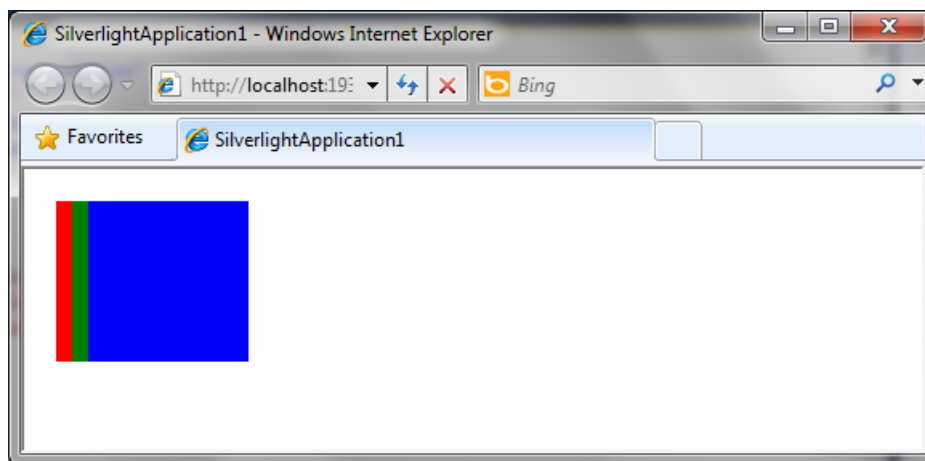
</UserControl>
```



2-12 ábra: A Canvas használata

Amikor a **Canvas** vezérlővel dolgozunk, megadhatunk minden vezérlő számára egy **ZIndex** értéket, amely a mélységi elhelyezkedést szabályozza, vagyis azt, hogy egy adott vezérlő eltakarhat-e egy másikat. Minél magasabb ez az érték, annál „közelebb” van a felszínhez az elem. Ha nem adtunk meg **ZIndexet**, akkor a definíciók sorrendjében jelennek meg a vezérlők, az először hozzáadott lesz legalul, amint azt a 2-13 ábra szemlélteti.

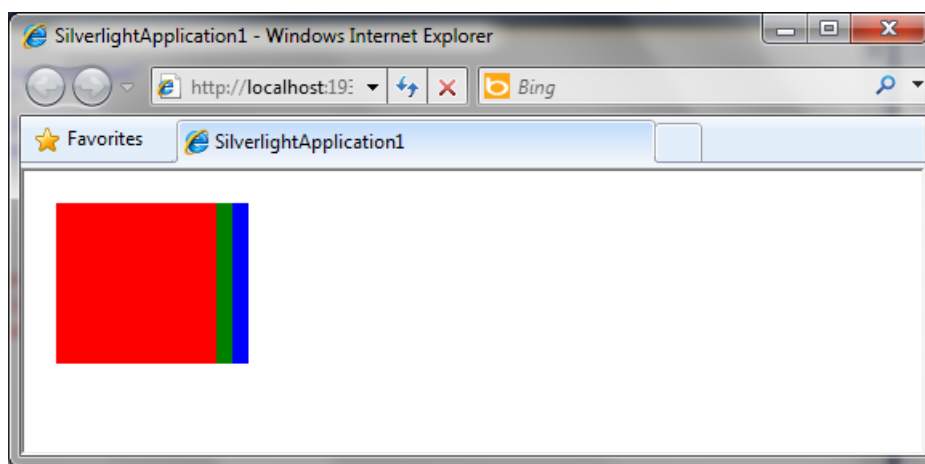
```
<Canvas x:Name="LayoutRoot">
    <Rectangle Width="100" Height="100" Fill="Red" Canvas.Left="20" Canvas.Top="20" />
    <Rectangle Width="100" Height="100" Fill="Green" Canvas.Left="30" Canvas.Top="20" />
    <Rectangle Width="100" Height="100" Fill="Blue" Canvas.Left="40" Canvas.Top="20" />
</Canvas>
```

2-13 ábra: Egymást elfedő vezérlőelemek

Módosítsuk a definíciót, ennek hatását a 2-14 ábrán láthatjuk:

```
<Canvas x:Name="LayoutRoot">
  <Rectangle Width="100" Height="100" Fill="Red" Canvas.Left="20" Canvas.Top="20"
Canvas.ZIndex="3" />
  <Rectangle Width="100" Height="100" Fill="Green" Canvas.Left="30" Canvas.Top="20"
Canvas.ZIndex="2" />
  <Rectangle Width="100" Height="100" Fill="Blue" Canvas.Left="40" Canvas.Top="20"
Canvas.ZIndex="1" />
</Canvas>
```



2-14 ábra: A mélységi sorrend megváltozott

StackPanel

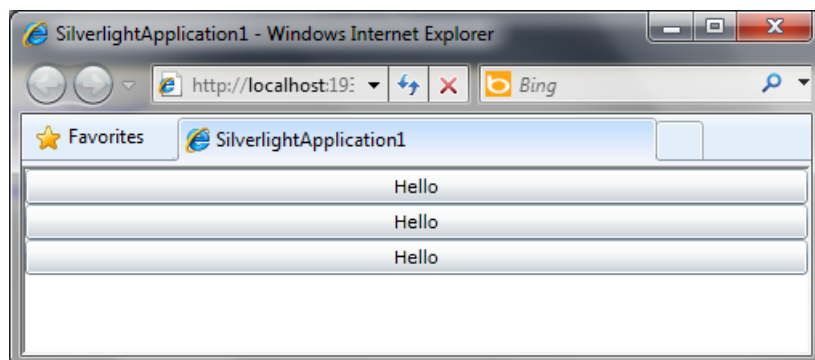
A **StackPanel** segítségével vízszintes vagy függőleges sorba rendezhetjük a vezérlőinket. Az alábbi definíció eredménye a 2-15 ábrán tekinthető meg.

```
<UserControl x:Class="SilverlightApplication1.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d"
d:DesignHeight="300" d:DesignWidth="400">

  <StackPanel x:Name="LayoutRoot">
```

```
<Button Content="Hello" />
<Button Content="Hello" />
<Button Content="Hello" />
</StackPanel>

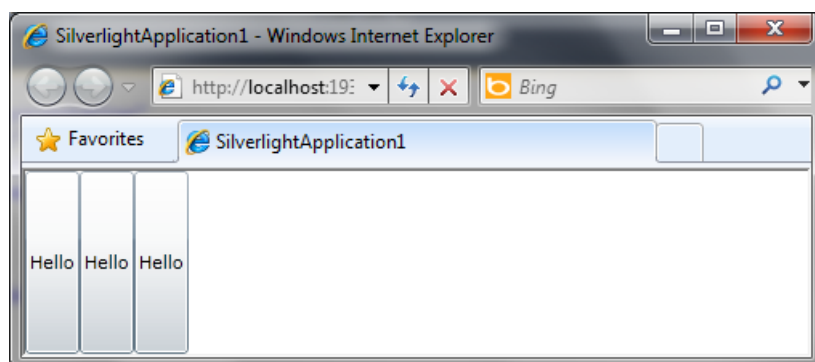
</UserControl>
```



2-15 ábra: A StackPanel használata

Az **Orientation** tulajdonsággal vízszintes irányba állíthatjuk az elemeket (2-16 ábra):

```
<StackPanel x:Name="LayoutRoot" Orientation="Horizontal">
```



2-16 ábra: A StackPanel elemeinek vízszintes elrendezése

Két olyan fontos dolog van, amit ennek a vezérlőnek a használata során tudnunk kell. Az első, hogy minden esetben egyenes sort kapunk, tehát ha nincs elég hely adott irányban, akkor nem folyik át a tartalom a következő sorba. A másik, hogy nem alkalmazkodik a böngésző méretéhez, azaz az esetlegesen túlnyúló részeket nem fogjuk viszontlátni.

WrapPanel

Ez a vezérlő nem része az alapsomagnak, a Silverlight Toolkit keretében juthatunk hozzá.

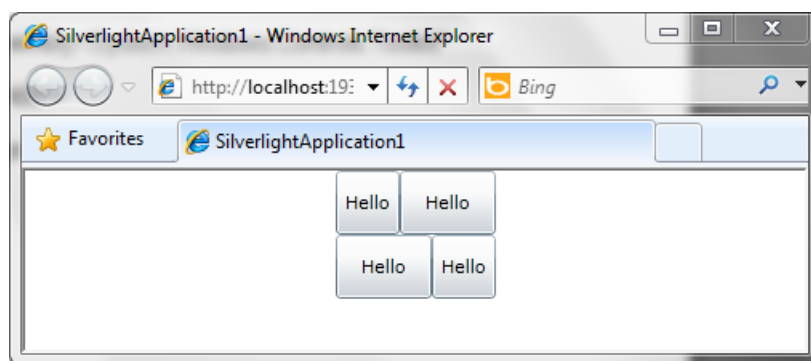
A Silverlight Toolkit telepítéséről illetve használatáról a következő, Vezérlők című fejezetben található részletes leírás.

Hasonló szerepet tölt be, mint a **StackPanel**, viszont van egy apró különbség, mégpedig az, hogy a lehető legoptimálisabban kitölti a rendelkezésre álló helyet, vagyis a vezérlők több sorban is megjelenhetnek. Az alábbi definíció eredménye a 2-17 ábrán látható:

```

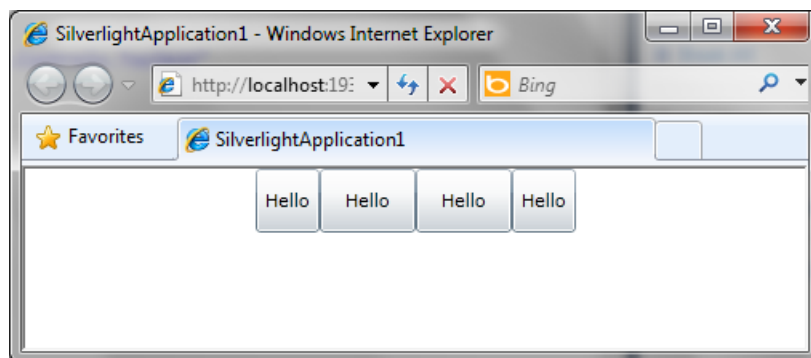
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:toolkit="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Toolkit"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="400">
  <StackPanel>
    <toolkit:WrapPanel Width="100" Height="200">
      <Button Width="40" Height="40" Content="Hello" />
      <Button Width="60" Height="40" Content="Hello" />
      <Button Width="60" Height="40" Content="Hello" />
      <Button Width="40" Height="40" Content="Hello" />
    </toolkit:WrapPanel>
  </StackPanel>
</UserControl>

```



2-17 ábra: A WrapPanel használata

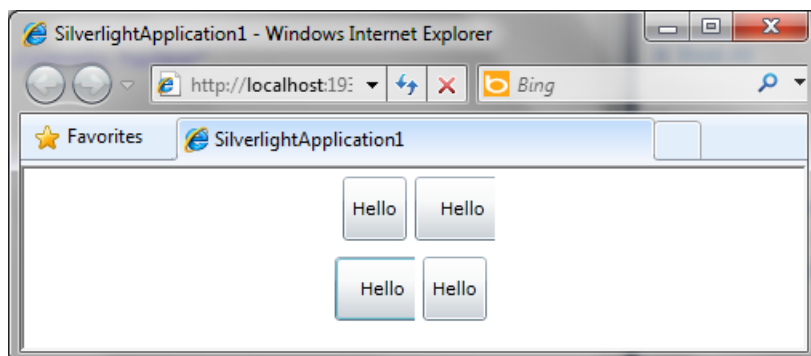
A fenti esetben két sorba rendezte a **WrapPanel** a vezérlőket, most cseréljük meg a szélesség és magasság értékeket és nézzük meg, hogy mit kapunk (2-18 ábra)!



2-18 ábra: A vezérlők egy sorban

Lehetőségünk van arra is, hogy minden elem számára uniform méretet adjunk meg, amely akkor is érvényes marad, ha a vezérlőnek így kevés hely jut. Erre a célra az **ItemHeight** és **ItemWidth** tulajdonságokat használhatjuk. A következő kód eredményét a 2-19 ábra mutatja:

```
<toolkit:WrapPanel ItemWidth="50" ItemHeight="50" Width="100" Height="200">
  <Button Width="40" Height="40" Content="Hello" />
  <Button Width="60" Height="40" Content="Hello" />
  <Button Width="60" Height="40" Content="Hello" />
  <Button Width="40" Height="40" Content="Hello" />
</toolkit:WrapPanel>
```

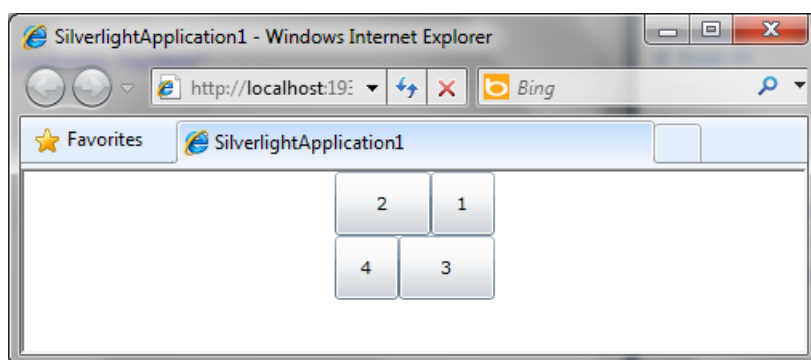


2-19: Az *ItemWidth* és *ItemHeight* használata

A **FlowDirection** tulajdonsággal a vezérlőelemek megjelenítésének sorrendjét szabályozhatjuk. Egyre azonban figyelniünk kell: ez csak soronként érvényes, nem globálisan, vagyis ha egy elem eredetileg az utolsó sor utolsó tagja lenne, az nem kerülhet fel az első sorba. A következő definíció erre mutat példát:

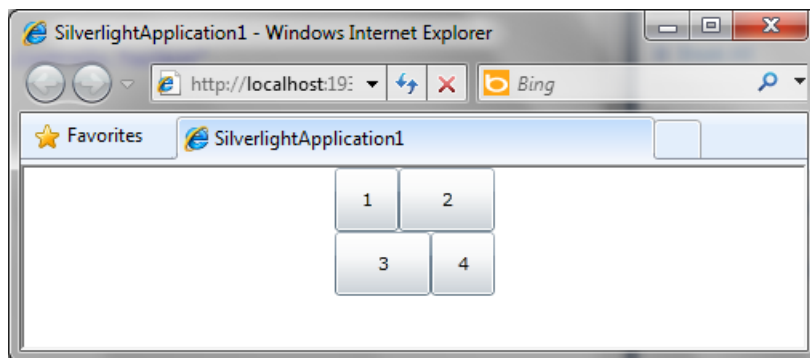
```
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:toolkit="clr-
namespace:System.Windows.Controls;assembly=System.Windows.Controls.Toolkit"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="400">
  <StackPanel>
    <toolkit:WrapPanel FlowDirection="RightToLeft" Width="100" Height="200">
      <Button Width="40" Height="40" Content="1" />
      <Button Width="60" Height="40" Content="2" />
      <Button Width="60" Height="40" Content="3" />
      <Button Width="40" Height="40" Content="4" />
    </toolkit:WrapPanel>
  </StackPanel>
</UserControl>
```

A 2-20 ábrán pedig az eredményt láthatjuk.



2-20 ábra: A *FlowDirection* használata

A **FlowDirection** nélküli – eredeti – alkalmazást a 2-21 ábrán tekinthetjük meg.



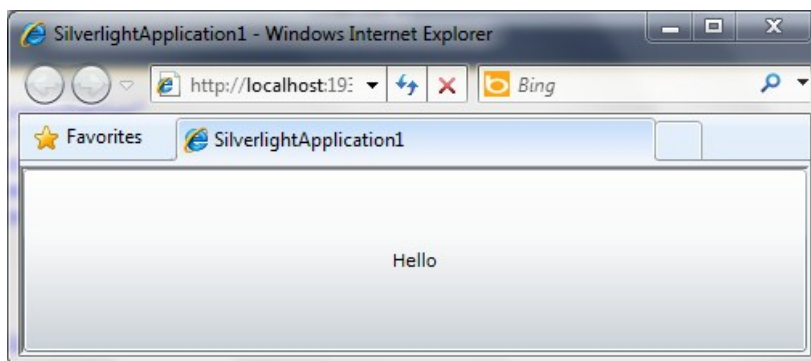
2-21 ábra: FlowDirection nélkül

DockPanel

Szintén a Silverlight Toolkitben megvalósított konténerrel van dolgunk a **DockPanel** személyében, amelynek segítségével a vezérlőelemeket egy-egy oldalhoz „ragaszthatjuk”. Nézzünk egy egyszerű példát, amely eredményét a 2-22 ábra mutatja be:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:toolkit="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Toolkit"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="400">

  <toolkit:DockPanel>
    <Button toolkit:DockPanel.Dock="Top" Content="Hello" />
  </toolkit:DockPanel>
</UserControl>
```

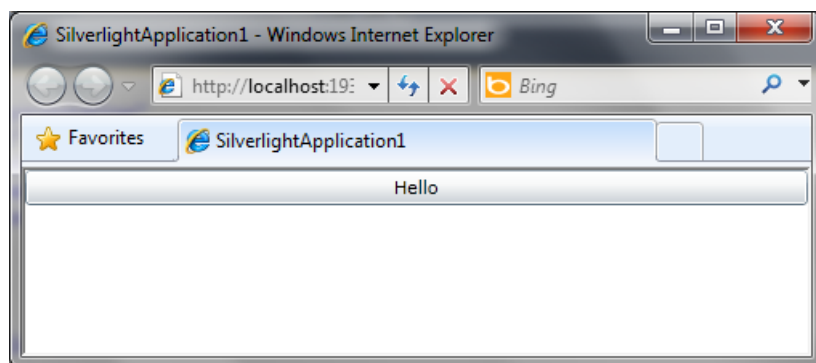


2-22 ábra: Egy gomb a DockPanel-ben

Valószínűleg senki nem erre számított, mi történt? A **DockPanel.Dock** attached propertyvel a konténer felső részéhez szegeztük a gombot, de azt nem mondtuk meg, hogy mennyi helyet foglalhat. Alapértelmezés szerint ilyenkor a teljes teret betölti a hozzáadott elem, ezen kell módosítanunk:

```
<toolkit:DockPanel LastChildFill="False">
  <Button toolkit:DockPanel.Dock="Top" Content="Hello" />
</toolkit:DockPanel>
```

A kiemelt tulajdonság gondoskodik arról, hogy az utoljára hozzáadott vezérlő ne akarjon terjeszkedni. Az eredményt a 2-23 ábra mutatja.

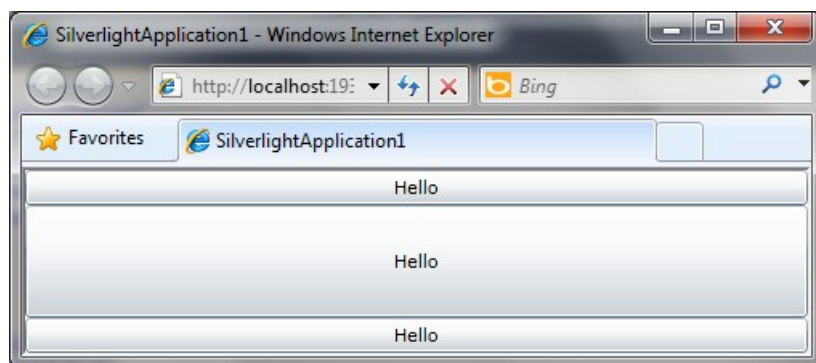


2-23 ábra: A LastChildFill tulajdonság használata

Vegyük észre, hogy ez a tiltás csak a függőleges dimenzióra vonatkozik, szélkében a nyomógomb továbbra is kövér marad.

A **LastChildFill** egy érdekes felhasználását vizsgáljuk meg a következő példában (2-24 ábra):

```
<toolkit:DockPanel LastChildFill="True">
  <Button toolkit:DockPanel.Dock="Top" Content="Hello" />
  <Button toolkit:DockPanel.Dock="Bottom" Content="Hello" />
  <Button Content="Hello" />
</toolkit:DockPanel>
```

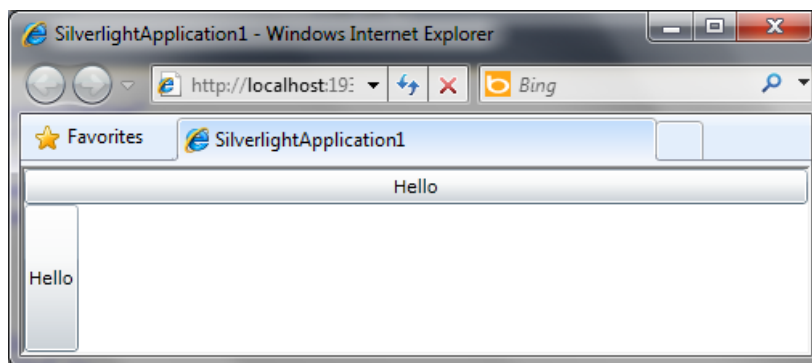


2-24 ábra: Több nyomógomb a DockPanelben

A vezérlőket a rendszer érzékelési sorrendben jeleníti meg, vagyis az utoljára hozzáadott elem most az első két gomb közé került.

Az elemek sorrendjéhez kapcsolódó kérdés: mi történik, ha egy vezérlőt felülre, egyet pedig balra pozicionálunk? Elvileg az utoljára érkezőnek takarnia kellene az elsőt, de szerencsére nem ez a helyzet (2-25 ábra):

```
<toolkit:DockPanel LastChildFill="False">
  <Button toolkit:DockPanel.Dock="Top" Content="Hello" />
  <Button toolkit:DockPanel.Dock="Left" Content="Hello" />
</toolkit:DockPanel>
```



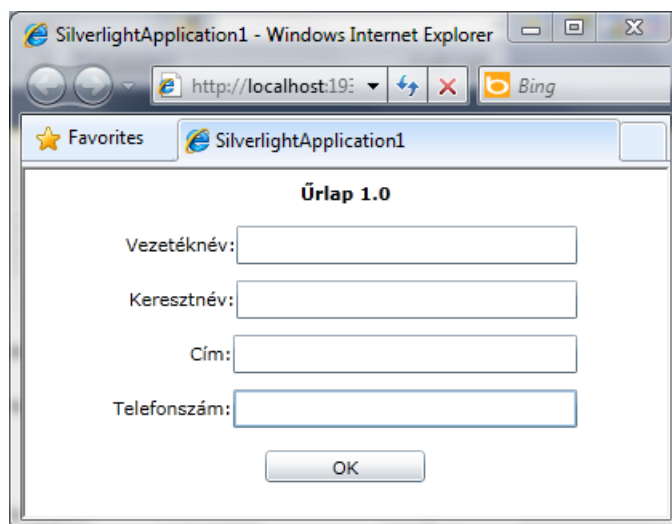
2-25 ábra: Különböző oldalakra ragasztott vezérlőelemek a DockPanelben

Látható, hogy a korábban definiált vezérlőnek több hely jutott, vagyis figyelniük kell a vezérlők sorrendjére.

Felület kialakítása konténerek egymásba ágyazásával

Csupán egyetlen konténer felhasználásával nehézkesen tudjuk egy alkalmazás teljes felületét elkészíteni. A **Grid** ugyan rugalmas, de túlzott használatával hosszú és kevésbé áttekinthető definíciót kapunk.

Ebben a fejezetben lépésről lépésre elkészítünk egy egyszerű űrlapot, a kész alkalmazást a 2-26 ábra mutatja.

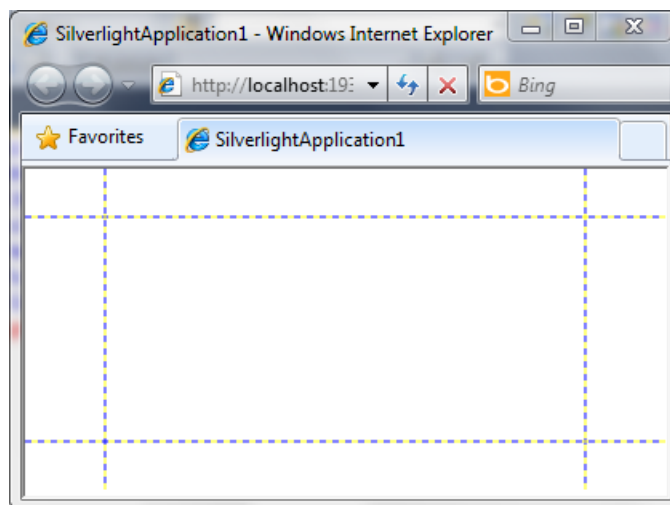


2-26 ábra: A kész űrlap

A tervezőnézetet használva ezt a felületet néhány perc alatt össze lehet „kattintgatni”, de ekkor számolnunk kell a forráskód elburjánzásával. Ehelyett egy elegáns, könnyen módosítható definíciót szeretnénk leírni, amely minél kevesebb rögzített adatot tartalmaz.

Az esetek igen nagy többségében a felület alapjának célszerű a **Grid** konténert választani, amelynek segítségével kialakítható az alkalmazás váza. A példában sincs ez másképp, lássuk, hogyan osztottuk fel a rendelkezésünkre álló területet! Alapvetően három szekciót tudunk megkülönböztetni: egy fejléct (Űrlap 1.0 szöveg), az adatbekérő mezőket, illetve a felület alján a gombot, amely az adatok beküldésére szolgál. Tehát három sort kell definiálnunk, de mi a helyzet az oszlopokkal? Több lehetőségünk is van, használhatjuk a **Vertical/HorizontalAlignment** párost, de akár a **Grid** oszlopdefinícióival is megoldhatjuk az elrendezést. Használjuk most ez utóbbit, így egy 3x3 –as rácsot kapunk, ezt a következő XAML mutatja (a végeredményt a 2-27 ábrán láthatjuk):

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    xmlns:toolkit="
        clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Toolkit"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="300" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="30" />
            <RowDefinition Height="*" />
            <RowDefinition Height="30" />
        </Grid.RowDefinitions>
    </Grid>
</UserControl>
```



2-27 ábra: Az űrlap váza

A középső oszlop, amely majd a tényleges űrlapot tartalmazza, minden esetben a böngészőablak közepén foglal helyet a középső, fix szélességű oszlopban.

A következő döntésünk arra vonatkozik, hogy miképpen rendezzük el az űrlap elemeit. Átgondolva a dolgot megállapíthatjuk, hogy az egyes mezőket a címkékkel egységként kezelhetjük, például egy **StackPanel** –be helyezve.

Ezzel analóg módon az összes címke-mező párost szintén egy **StackPanel**-be tölthetjük, függőleges orientációval. A következő XAML kódban ezt valósítjuk meg, az eredményt a 2-28 ábra mutatja:

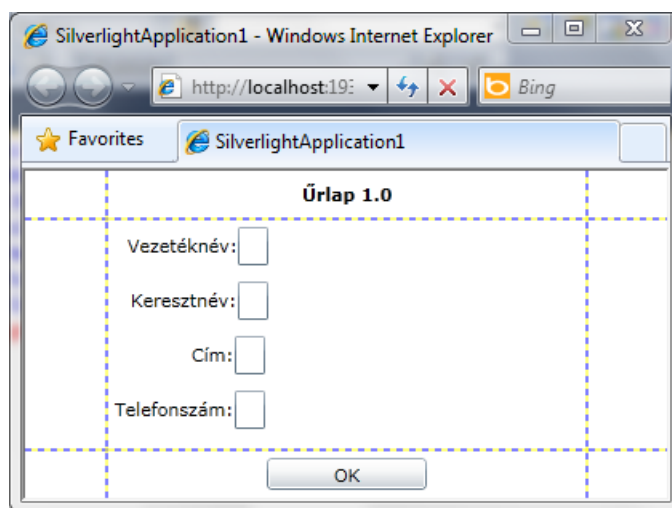
```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    xmlns:toolkit="clr-namespace:System.Windows.Controls;
        assembly=System.Windows.Controls.Toolkit"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid ShowGridLines="True">
```



```

<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="300" />
    <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="30" />
    <RowDefinition Height="*" />
    <RowDefinition Height="30" />
</Grid.RowDefinitions>
<TextBlock FontWeight="Bold" Text="Űrlap 1.0" VerticalAlignment="Center"
    HorizontalAlignment="Center"
    Grid.Column="1" Grid.Row="0" />
<StackPanel Orientation="Vertical" Grid.Column="1" Grid.Row="1">
    <StackPanel Orientation="Horizontal" Margin="5">
        <TextBlock Text="Vezetéknév:" VerticalAlignment="Center"
            Margin="8,0,0,0" />
        <TextBox />
    </StackPanel>
    <StackPanel Orientation="Horizontal" Margin="5">
        <TextBlock Text="Keresztnév:" VerticalAlignment="Center"
            Margin="10,0,0,0" />
        <TextBox />
    </StackPanel>
    <StackPanel Orientation="Horizontal" Margin="5">
        <TextBlock Text="Cím:" VerticalAlignment="Center" Margin="48,0,0,0" />
        <TextBox />
    </StackPanel>
    <StackPanel Orientation="Horizontal" Margin="5">
        <TextBlock Text="Telefonszám:" VerticalAlignment="Center" />
        <TextBox />
    </StackPanel>
</StackPanel>
<Button Content="OK" Margin="100,5" Grid.Column="1" Grid.Row="2" />
</Grid>
</UserControl>

```



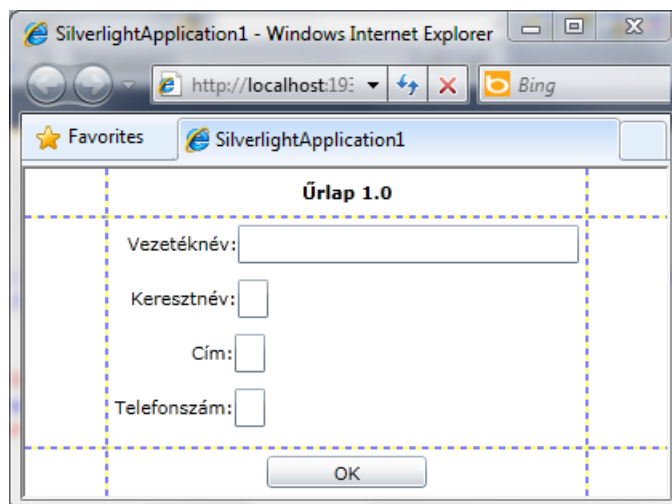
2-28 ábra: Egymásba ágyazott StackPanel-ek

A címkéket a **Margin** tulajdonság használatával jobbra, pontosabban a leghosszabb „Telefonszám” mezőhöz igazítottuk (ez nyilván ízlés dolga, többféle megoldás létezik).

Látható, hogy a **TextBox**-ok nem úgy viselkednek, mint amire számítottunk. Ennek oka nagyon egyszerű: mivel nem adtunk meg külön szélesség értéket, ezért a **StackPanel** a lehető legkisebb helyfoglalással jeleníti meg a vezérlőt.

Két megoldás van a problémára, az első a legegyszerűbb: beállítjuk a **width** tulajdonságot. Ekkor viszont számolnunk kell azzal, hogy az űrlap módosításakor a rendelkezésünkre álló hely csökkenhet, vagyis ilyenkor a vezérlők méretét is meg kell változtatni. A másik lehetőségünk, hogy a **StackPanel** helyett a **DockPanel** vezérlőt használjuk, és a **LastChildFill** tulajdonságot hívjuk segítségül, azaz a **TextBox** mérete alkalmazkodni fog az őt tároló konténer méretéhez. Ez utóbbi esetben a definíció és a felület (2-29 ábra) a következőképpen alakul:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    xmlns:toolkit="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Toolkit"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="300" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="30" />
            <RowDefinition Height="*" />
            <RowDefinition Height="30" />
        </Grid.RowDefinitions>
        <TextBlock FontWeight="Bold" Text="Űrlap 1.0" VerticalAlignment="Center"
            HorizontalAlignment="Center"
            Grid.Column="1" Grid.Row="0" />
        <StackPanel Orientation="Vertical" Grid.Column="1" Grid.Row="1">
            <toolkit:DockPanel LastChildFill="True" Margin="5">
                <TextBlock Text="Vezetéknév:" VerticalAlignment="Center"
                    Margin="8,0,0,0" />
                <TextBox />
            </toolkit:DockPanel>
            <StackPanel Orientation="Horizontal" Margin="5">
                <TextBlock Text="Keresztnév:" VerticalAlignment="Center"
                    Margin="10,0,0,0" />
                <TextBox />
            </StackPanel>
            <StackPanel Orientation="Horizontal" Margin="5">
                <TextBlock Text="Cím:" VerticalAlignment="Center" Margin="48,0,0,0" />
                <TextBox />
            </StackPanel>
            <StackPanel Orientation="Horizontal" Margin="5">
                <TextBlock Text="Telefonszám:" VerticalAlignment="Center" />
                <TextBox />
            </StackPanel>
        </StackPanel>
        <Button Content="OK" Margin="100,5" Grid.Column="1" Grid.Row="2" />
    </Grid>
</UserControl>
```



2-29 ábra: A DockPanel-t használtuk

Ezzel el is készültünk az űrlappal. Egy ilyen egyszerű példán keresztül is jól látható, hogy konténerek egymásba ágyazásával milyen könnyű rugalmas, a böngésző méretváltozásaihoz alkalmazkodó felületet készíteni.

Összefoglalás

Ebben a fejezetben bepillantást nyerhettünk a felület megjelenítésének életciklusába, illetve megismerkedtünk a négy konténelemelemmel, amelyek a felület határvonalait definiálják.

A Microsoft mind a WPF, mind a Silverlight esetében szakított a hagyományokkal, ennek köszönhetően az egyszerűbben használható, ám kevésbé rugalmas koordináta alapú felhasználói felület helyett egy komplexebb, viszont alkalmazkodóbb és könnyen karbantartható kódot eredményező eszközt kaptunk.

Láthattuk, hogy a megoldáshoz többféleképpen közelíthetünk, akár egyetlen konténertípussal, akár több egymásba ágyazásával.

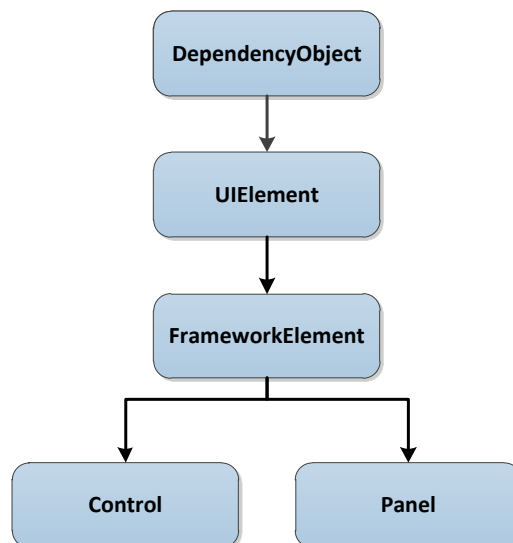
3. Alapvető vezérlők

Vezérlők alatt azokat a — legtöbbször vizuális — elemeket értjük, amelyek segítségével a felhasználó az alkalmazással kommunikál. Ebben a fejezetben megismerkedünk a vezérlők hierarchiájával, és a leggyakrabban használt elemekkel.

Osztályhierarchia

A legtöbb fejlesztői környezetben egyértelműen azonosíthatóak a vezérlőelemek: ezek azok, amelyek egyrészt vizuálisan megjelennek, másrészt a felhasználó közvetlen interakciót végezhet velük, például rájuk kattinthat.

A Silverlight osztályhierarchiája (3-1 ábra) kevésbé teszi egyértelművé ezt a helyzetet, mivel sok olyan elemet is kezelhetünk vezérlőként, amelyek valójában nem azok.

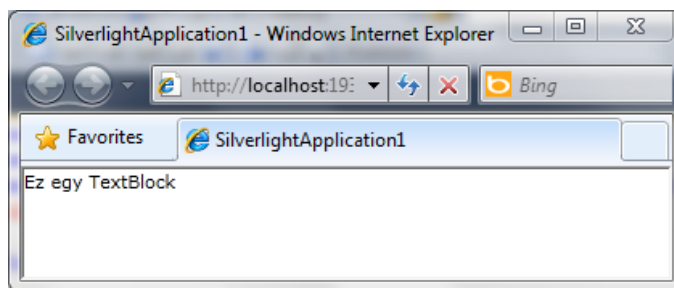


3-1 ábra: Silverlight osztályhierarchia

A hierarchia legalján a **DependencyObject** absztrakt osztály áll, ő gondoskodik többek között az adatkötésekről és a tulajdonságrendszeréről is (*dependency property*, *attached property*). Ez az osztály nem kizárólagosan a vezérlő osztályok öse, lényegében a legtöbb Silverlight osztály belőle származik.

A második szinten az **UIElement** áll. Ez felel a vizuális megjelenítésért, illetve kezeli a felhasználói interakciókat (input). Szintén hozzá tartoznak az *Arrange/Measure* metódusok is.

A **UIElement**-ből származó **FrameworkElement** nem tesz mást, mint kiterjeszti a fölötte lévő képességeit. Az teszi érdekessé ezt az osztályt, hogy a belőle származók már teljes értékű vezérlőknek tekinthetők, viszont a gyakorlatban nem azok. A legjobb példa erre a **TextBlock** osztály (3-2 ábra), amelynek bár van vizuális megjelenése és képes fogadni a felhasználótól érkező utasításokat, mégsem „igazi” vezérlő.



3-2 ábra: TextBlock „vezérlő”

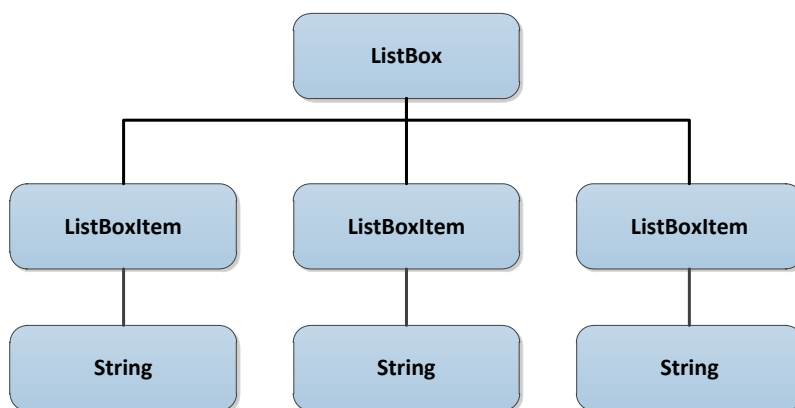
Vegyük észre, hogy a **TextBlock**-ra nem hatnak alapértelmezetten a **VerticalAlignment** és **HorizontalAlignment** tulajdonságok! Ezek ugyanis csak a **Control** osztályból közvetlenül származó elemekre érvényesek – annak ellenére, hogy ezt a két tulajdonságot a **FrameworkElement** szolgáltatja.

A **FrameworkElement**-ből származik a **Panel** osztály is, amely az összes konténerelem őse.

A hierarchia legalján álló **Control** osztály az „igazi” vezérlők őse. Valójában egyetlen (ugyanakkor nagyon fontos) említésre méltó szolgáltatással egészíti ki a listát, mégpedig a sablonokkal. A Silverlight ún. „lookless” megközelítést alkalmaz, vagyis a vezérlők megjelenését és funkcionalitását teljesen szétválasztja. Ebből a filozófiából alakult ki a „parts and states” modell is, amely „elvárja”, hogy egy vezérlő szolgáltatssa a működéséhez szükséges részegységeket (*parts*), viszont ezek megjelenését szabadon variálhatjuk (*states*).

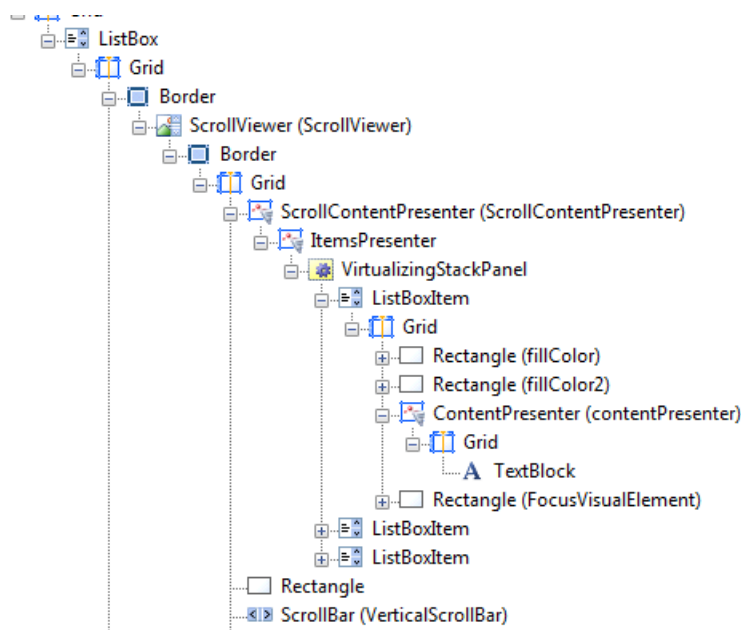
Logikai és vizuális fa

Minden vezérlő egy sablon alapján épül fel, ezt kétféleképpen közelíthetjük meg. Vegyük például a **ListBox** vezérlőt! Ez az elem **ListBoxItem** példányokat tárol, amelyek számunkra string típusú elemekként jelennek meg, vagyis gyakorlati szempontból a következőt látjuk:



3-3 ábra: ListBox logikai fa

Ugyanakkor a valóságban a **ListBox** apró – vizuális – egységekre bontható, ahogyan azt a 3-4 ábra is mutatja (a kép a Silverlight Spy programból származik).



3-4 ábra: ListBox vizuális fa

Az első esetet a vezérlő logikái, míg a másodikat vizuális fájának nevezzük.

Dependency property , Attached property

A vezérlők tulajdonságai számunkra hagyományos tulajdonságként jelennek meg, de a valóságban ezek mögött egy speciális típus, az ún. *dependency property* rejtőzik. Ezek jelentősége abban rejlik, hogy értékük a rendszer több eleméből számolódik, ilyenek például a stílusok, adatkötések, animációk vagy egy, az objektumfában feljebb lévő elem is.

A következő definícióban egy gomb háttérszínét állítjuk be:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
  mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
  <Grid>
    <Button Background="Red" Content="Gomb" />
  </Grid>
</UserControl>
```

Lássuk, hogy mi történik a valóságban! A **Button** osztálynak nincs **Background** tulajdonsága, ehelyett az öröklési fán feljebb kell lépnünk, egészen a **Control** osztályig. Itt már megtaláljuk ezt a tulajdonságot, a Reflectort felhasználva meg tudjuk nézni, hogy mi történik (3-5 ábra).

```
public Brush Background
{
    get
    {
        return (Brush) base.GetValue(BackgroundProperty);
    }
    set
    {
        base.SetValue(BackgroundProperty, (DependencyObject) value);
    }
}
```

3-5 ábra A *Background* tulajdonság mögötti kód

Látható, hogy ez a tulajdonság csak burkolásként szolgál az „igazi” **BackgroundProperty** számára, amelynek a definíciója a következőképpen néz ki:

```
public static readonly DependencyProperty BackgroundProperty;
```

A dependency propertyk neve konvenció szerint a **Property** utótagot kapja. A **GetValue** és **SetValue** metódusok a **DependencyObject** osztályból származnak, minden érték rajtuk megy keresztül, mire eljut a vezérlőkhöz.

Mivel több adatforrás létezhet egy időben, ezért köztük precedencia létezik, amelynek első helyén az animációk állnak, őket követik az adatkötések, sablonok, stílusok, végül pedig az alapértelmezett értékek.

A dependency propertyk egy speciális típusa az *attached property*, amely lehetővé teszi érték hozzárendelését egy tulajdonsághoz olyan objektumon, amely igazából nem rendelkezik vele. Tipikusan olyan helyzetben használjuk, ahol valamiféle szülő-gyermek kapcsolat áll fenn, például ilyen a **Grid** konténer **Row** és **Column** tulajdonsága is, amely a tartalmazott elem helyzetét jelöli a **Grid**-en belül.

Eseménykezelés

A Silverlight eseményei hagyományos CLR (Common Language Runtime) események, amelyek egyaránt deklarálhatóak a felhasznált programozási nyelven és közvetlenül az XAML definícióban is. A következő forráskód az első lehetőséget mutatja be, feltételezve, hogy a felületen létezik egy **button1** nevű nyomógomb:

```
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;

namespace SilverlightApplication1
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
            button1.Click += new RoutedEventHandler(button1_Click);
        }

        void button1_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Click");
        }
    }
}
```


Látható, hogy a szokásos módszerrel rendelhetünk eseménykezelőt az eseményekhez, viszont ezt minden esetben az **InitializeComponent** metódus meghívása után kell megtennünk, mivel a gomb objektum csak ekkortól létezik (ellenkező esetben **NullReferenceException** kivételt kapunk az alkalmazás futtatásakor). Erre a célra a legmegfelelőbb a **UserControl Loaded** eseménye, mivel ez jelzi, hogy a vezérlő minden eleme betöltődött.

Elegánsabb és biztonságosabb megoldás, ha rögtön az XAML definícióban végezzük el a hozzárendelést, ezt az alábbi kódrészlet mutatja:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid>
        <Button x:Name="button1" Content="Click" Click="button1_Click" />
    </Grid>
</UserControl>
```

Ha ezt a módszert választjuk, akkor a **button1_Click** metódus automatikusan legenerálódik a XAML definícióhoz tartozó kódfájlban.

Vegyük észre, hogy egy új típusú eseménykezelőt — ez a **RoutedEventHandler** — használtuk a fenti kódokban! A Silverlight néhány eseménye ún. *routed event*, vagyis olyan események, amelyeket nem muszáj helyben kezelni, mivel képesek „felfelé vándorolni” a vizuális fában.

A **RoutedEventArgs** osztály **OriginalSource** paraméter visszaadja az eseményt küldő elemet, de nem a vezérlőt, amelyen eredetileg kezeltük, hanem annak komponensét. Például, ha egy gombon a felíratra kattintunk, akkor az azt tároló **TextBlock** objektumra mutat az **OriginalSource**.

Vezérlők a gyakorlatban

Ebben a fejezetben megismerkedünk a Silverlight fontosabb vezérlőivel. Mielőtt ezt megtennénk, telepítenünk kell a Silverlight Toolkitet is, amely az alapértelmezett vezérlőket továbbiakkal egészíti ki.

A Toolkitet a <http://silverlight.codeplex.com/> oldalról tölthetjük le, a telepítőkészletes változatot érdemes választani. Az assemblyket a **Program Files\Microsoft SDKs\Silverlight\v4.0\Toolkit** könyvtárba pakolja a telepítő. Ebben a fejezetben a **System.Windows.Controls.Toolkit.dll** fájlra lesz majd szükségünk, ezt kell hozzáadni a project referenciáihoz.

Még egy utolsó lépést kell tennünk: az XAML file fejlécében található néhány **xmlns** kezdetű sor, amelyek névttereket jelölnek, ezekhez kell hozzáadni a Toolkit elérhetőségét is. A művelethez IntelliSense segítséget is kapunk:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    xmlns:toolkit=
        "clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Toolkit"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid>

    </Grid>
</UserControl>
```

3. Alapvető vezérlők

Az **xmlns** után meg kell adnunk egy előtagot, amelynek segítségével a definícióban hivatkozhatunk az adott névtérre, ez tetszőleges, a példában most **toolkit** lesz. Lássuk, hogyan használhatjuk ezt, cseréljük a **Grid**-et **DockPanel**-re:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
  xmlns:toolkit="clr-namespace:System.Windows.Controls;
  assembly=System.Windows.Controls.Toolkit"
  mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
  <toolkit:DockPanel>

  </toolkit:DockPanel>
</UserControl>
```

Előfordulhat, hogy a **toolkit** prefix után nem működik az IntelliSense, ez egy ismert és eddig még nem javított hiba. Meg lehet próbálni eltávolítani, illetve újra hozzáadni a Toolkit referenciát, esetleg a Silverlight Tools eltávolítása vagy újrategyűjtése is működhet, de sajnos egyik megoldás sem működik biztosan.

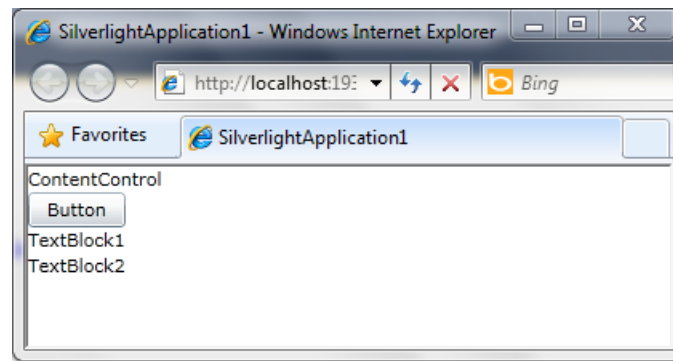
ContentControl

A **ContentControl** osztályból származó vezérlők a **Content** tulajdonságukon keresztül jelenítenek meg tartalmat. Mivel ez a tulajdonság **object** típusú, ezért nagyjából bármit megadhatunk számára, ekkor az **UIElement**-ből származó osztályokat vizuális megjelenésük szerint, minden mást pedig a **ToString** metódussal kinyert stringként jelenít meg.

A **ContentControl** érdekes vonása, hogy önmagában is használható, erre a következő definíció mutat példát (az eredmény pedig a 3-6 ábrán látható):

```
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
  mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
  <StackPanel>
    <ContentControl Content="ContentControl" />

    <ContentControl>
      <ContentControl.Content>
        <StackPanel>
          <Button Content="Button" />
          <TextBlock Text="TextBlock1" />
          <TextBlock Text="TextBlock2" />
        </StackPanel>
      </ContentControl.Content>
    </ContentControl>
  </StackPanel>
</UserControl>
```



3-6 ábra: A ContentControl vezérlő

Látható, hogy a **Content** tulajdonság a definícióban kifejtve összetett objektumokat is képes fogadni.

Button, RepeatButton és HyperlinkButton

A legismertebb **ContentControl** vezérlők a különböző nyomógombvariánsok, ezek a **ButtonBase** osztályból származnak, amely pedig értelemszerűen a **ContentControl**-ből.

A **Button** vezérlő funkciójában nem különbözik a más keretrendszerekben találhatóaktól, illetve őt már használtuk korábbi fejezetekben, így nem vesztegetek rá több szót.

Annál érdekesebb a **RepeatButton**, amely lényegében nem különbözik egy hagyományos gombtól, egy apró különbséget kivéve: a nyomva tartásával folyamatosan küldi a **Click** eseményeket. A következő definíció erre mutat példát:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
  mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
  <StackPanel>
    <TextBlock x:Name="txtbox" Text="0" />
    <RepeatButton Content="Click" Click="Button_Click" />
  </StackPanel>
</UserControl>
```

A kapcsolódó kódfájl pedig ilyen lesz:

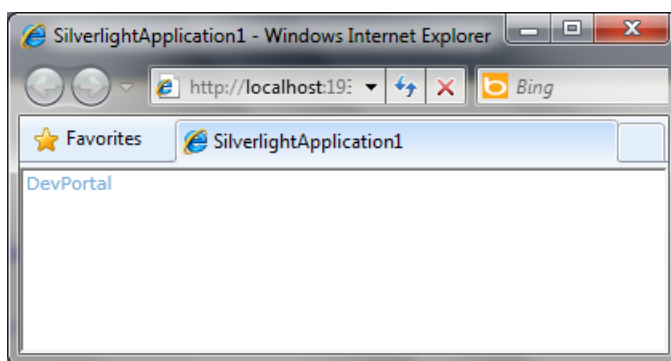
```
namespace SilverlightApplication1
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            txtbox.Text = (int.Parse(txtbox.Text) + 1).ToString();
        }
    }
}
```

Ha az alkalmazásban nyomva tartjuk a gombot, az folyamatosan növeli a megjelenített értéket.

A **HyperlinkButton** a nevéhez híven átjárót biztosít egy — az alkalmazás szempontjából — külső weboldalra vagy valamilyen tartalomra (3-7 ábra):

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <StackPanel>
        <HyperlinkButton Content="DevPortal" NavigateUri="http://www.devportal.hu"
            TargetName="_blank" />
    </StackPanel>
</UserControl>
```



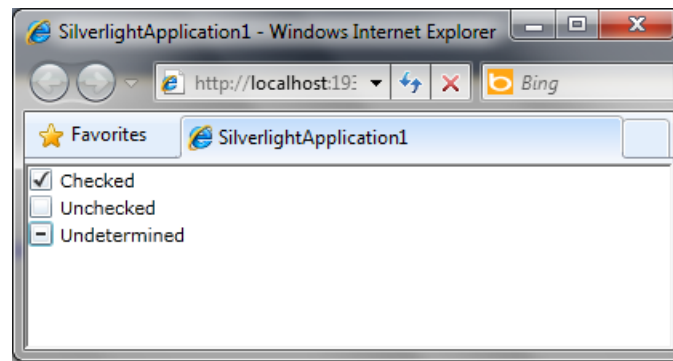
3-7 ábra: A **HyperLinkButton** vezérlő

CheckBox és RadioButton

Ez a két vezérlő a **ToggleButton** osztályból származik, amely kiegészíti a **ButtonBase** osztályt az állapotváltás képességével. A **ToggleButton** leszármazottaknak háromféle állapota lehet: jelölt (*checked*), nem jelölt (*unchecked*) és határozatlan (*undetermined*). A harmadik állás csak akkor létezik, ha az adott vezérlőn az **IsThreeState** tulajdonságot igaz értékre állítottuk. Az alapállapot minden esetben az *unchecked* lesz.

Elsőként lássuk, hogyan működik a **CheckBox**, a következő definíció eredményét a 3-8 ábra mutatja:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <StackPanel>
        <CheckBox x:Name="cb1" Content="Checked" IsChecked="True" IsThreeState="True" />
        <CheckBox x:Name="cb2" Content="Unchecked" IsChecked="False" IsThreeState="True"/>
        <CheckBox x:Name="cb3" Content="Undetermined" IsChecked="{x:Null}"
            IsThreeState="True" />
    </StackPanel>
</UserControl>
```



3-8 ábra: A CheckBox vezérlő három állása

Az **IsChecked** tulajdonság egy **Nullable<bool>** típust képvisel, így háromféle értéket vehet fel: **true**, **false** és **null** — ez a sorrend megfelel a **CheckBox** lehetséges állapotainak.

Az egyes állapotok külön eseménykezelővel rendelkeznek, ezek sorrendben: **Checked**, **Unchecked** és **Indeterminate**.

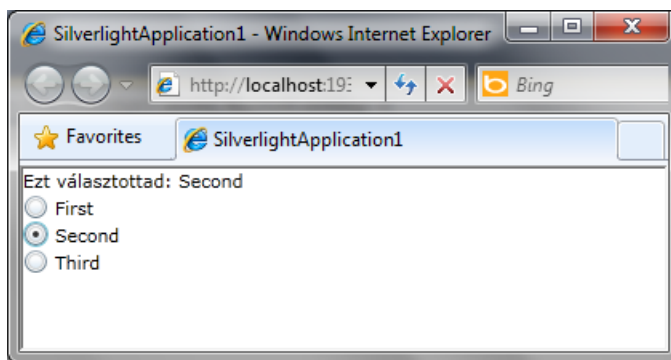
A **RadioButton** vezérlőt tipikusan csoportban használjuk, az egyes vezérlőket a **GroupName** tulajdonságon keresztül tudjuk egymáshoz rendelni. Az alábbi definíció és forráskód eredménye a 3-9 ábrán látható:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <StackPanel>
        <TextBlock x:Name="tb1" />
        <RadioButton GroupName="group1" Content="First" Checked="RadioButton_Checked" />
        <RadioButton GroupName="group1" Content="Second" Checked="RadioButton_Checked" />
        <RadioButton GroupName="group1" Content="Third" Checked="RadioButton_Checked" />
    </StackPanel>
</UserControl>
```

Mindhárom **RadioButton** ugyanazon az eseménykezelőn osztozik, a forráskód a következőképpen fest:

```
private void RadioButton_Checked(object sender, RoutedEventArgs e)
{
    var rb = sender as RadioButton;

    tb1.Text = "Ezt választottad: " + rb.Content;
}
```



3-9 ábra: A RadioButton vezérlő

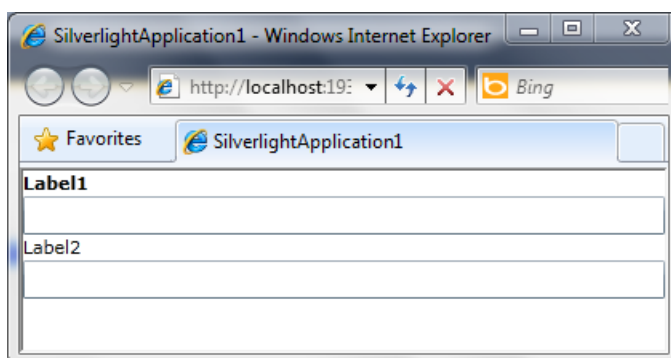
Label

A **Label**, bár funkciójában hasonlít a **TextBlock**-ra, annál jelentősen több lehetőséget biztosít. Mivel a **Label** valódi vezérlő, ezért tetszés szerint megváltoztathatjuk a megjelenését, de ami még ennél is fontosabb, a segítségével informatív űrlapokat hozhatunk létre, mivel az adatok érvényességét is képes jelezni.

Ez a vezérlő az SDK részét képezi, ez a Silverlight Tools telepítésekor automatikusan hozzáadódik a projektsablonhoz, mindössze az **sdk** prefixet kell használnunk.

A **Label** erősen épít az adatkötésekre, ezért egyelőre meg kell elégednünk egy egyszerűbb példával. A **Target** tulajdonság segítségével a **Label**-t egy másik (jellemzően input) vezérlőhöz köthetjük. A következő példában az **IsRequired** tulajdonságot fogjuk használni, ezzel jelezzük a **Label**-nek, hogy a hozzá kötött beviteli mezőt kötelező kitölteni, és így a **Label** félkövéren jelenik majd meg. A következő definíció eredményét a 3-10 ábrán tekinthetjük meg:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <StackPanel>
        <sdk:Label Content="Label1" Target="{Binding ElementName=tb1}" IsRequired="True" />
        <TextBox x:Name="tb1" />
        <sdk:Label Content="Label2" Target="{Binding ElementName=tb2}" />
        <TextBox x:Name="tb2" />
    </StackPanel>
</UserControl>
```



3-10 ábra: Label vezérlők

A **Target** beállításához egyszerű adatkötést használtunk, az **ElementName** tulajdonságnak a kiválasztott vezérlő nevét kell megadnunk. A **Label**-re nem kerülhet fókus, illetve a **TabStop** sem használható rajta. Bár rengeteg hasznos tulajdonsága van, a **Label** drága játékszer lehet, ha túlzásba vesszük használatát. Amikor csak és kizárólag szöveg megjelenítésére van szükség, akkor helyette célszerű a **TextBlock** elemet használni.

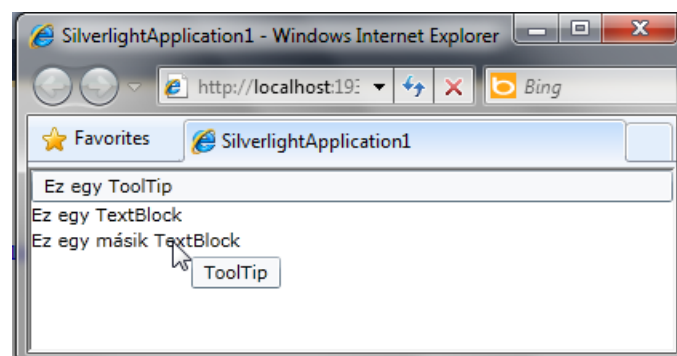
ToolTip

A **ToolTip** segítségével információt köthetünk egy-egy vezérlőelemhez, ezzel is segítve a felhasználót. Ez a vezérlő önmagában is használható, ekkor lényegében úgy viselkedik, mint egy **Label**. Leggyakrabban egy másik vezérlőhöz kötjük hozzá, a **ToolTipService** osztály tulajdonságaival. A következő definíció mindkét esetre mutat példát (az eredmény a 3-11 ábrán látható):

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <StackPanel>
        <ToolTip Content="Ez egy ToolTip" />

        <TextBlock Text="Ez egy TextBlock">
            <ToolTipService.ToolTip>
                <ToolTip Content="Ez pedig egy TextBlock-ToolTip" />
            </ToolTipService.ToolTip>
        </TextBlock>

        <TextBlock Text="Ez egy másik TextBlock"
            ToolTipService.Placement="Mouse"
            ToolTipService.ToolTip="ToolTip" />
    </StackPanel>
</UserControl>
```



3-11 ábra: A ToolTip vezérlő

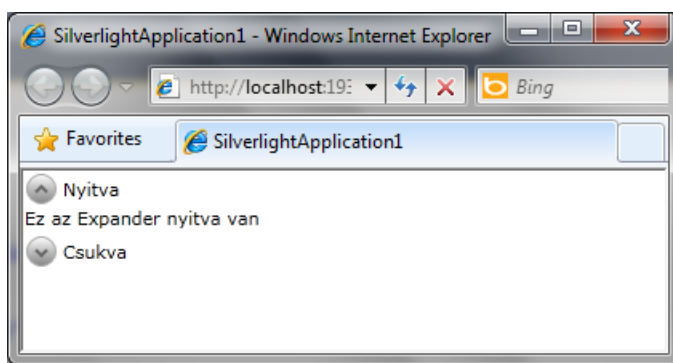
A **ToolTipService** objektum **Placement** tulajdonsága jelöli, hogy a **ToolTip** a vezérlőhöz képest milyen pozícióban jelenjen meg. A fent alkalmazott **Mouse** az egérmutató alá helyezi el. Amennyiben a **ToolTip** számára nincs elég hely a kijelölt pozíción, akkor a vezérlő jobb oldalán fog előtűnni.

Expander

Az **Expander** a **ContentControl** egy leszármazottjából, a **HeaderedContentControl**-ból származik, amely felruházta azzal a képességgel, hogy az eredeti tartalom mellett egy fejléct is képes megjeleníteni. Az **Expander** esetében a fő funkciót a tartalom elrejtése illetve megjelenítése jelenti, viszont a fejléc minden esetben látható marad.

Az **Expander** és a **HeaderedContentControl** is a Toolkit része. Az alábbi definíció eredményét mutatja be a 3-12 ábra.

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    xmlns:toolkit="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Toolkit"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <StackPanel>
        <toolkit:Expander Header="Nyitva" Content="Ez az Expander nyitva van" IsExpanded="True"
    />
        <toolkit:Expander Header="Csukva" Content="Ez az Expander csukva van" />
    </StackPanel>
</UserControl>
```



3-12 ábra: *Expander* vezérlők

Az **IsExpanded** tulajdonság segítségével szabályozhatjuk, hogy a vezérlő nyitva, illetve csukva van-e alapállapotában.

ItemsControl

Az **ItemsControl** osztályból származó vezérlők lehetővé teszik egyszerre több elem megjelenítését. Hasonlóan a **ContentControl**-hoz ez is használható önmagában, ekkor azonban le kell mondanunk az elemek kiválasztásának lehetőségéről.

Az **ItemsControls** leszármazottak az **ItemsSource** tulajdonságon keresztül kapják meg a megjelenítendő listára hivatkozó referenciát.

Az **ItemsSource** nem egyenértékű a **DataContext** tulajdonsággal, amelyet a **FrameworkElement**-től örököl a vezérlő. Ez utóbbi ugyanis általánosabb, a logikai fán keresztül látható adatforrás, míg az **ItemsSource** csak lokális, viszont képes megjeleníteni az elemeket, pontosabban elkészít számukra egy sablont.

Jellemzően a **DataContext**-et egy felső szintű – általában konténer – elemen használjuk, összetett adatforrást kötünk rá, és ennek egy részhalmazát adjuk meg egy **ItemsControl** leszármazott számára.

Az alábbi definíció és forráskód egy **ItemsSource** deklarációját és feltöltését mutatja be, az eredmény a 3-13 ábrán látható.


```

<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid>
        <ItemsControl x:Name="ic" />
    </Grid>
</UserControl>

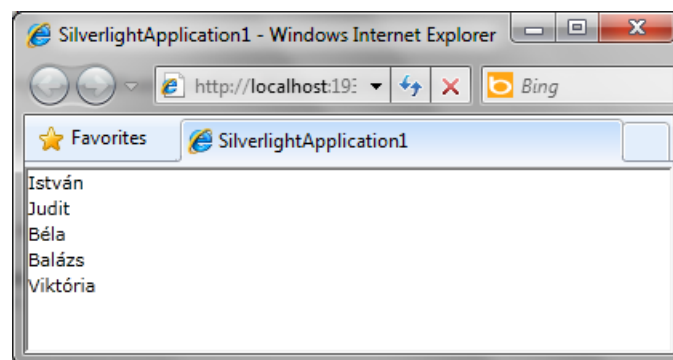
// --- Kódfájl

namespace SilverlightApplication1
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();

            List<string> nameList = new List<string>()
            {
                "István", "Judit", "Béla", "Balázs", "Viktória"
            };

            ic.ItemsSource = nameList;
        }
    }
}

```



3-13 ábra: Az ItemControl vezérlő

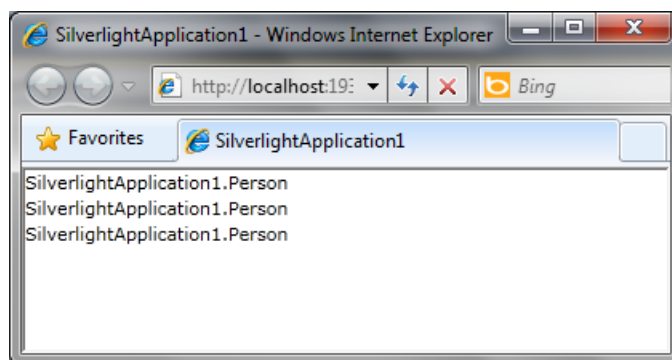
Összetett objektumok is lehetnek a lista elemei, ekkor meg kell adnunk azt a tulajdonságot, amelyet meg akarunk jeleníteni, ellenkező esetben az osztály neve jelenik meg (3-14 ábra). A következő forráskódban erre az esetre találunk példát (a 3-15 ábrán látható a jól működő alkalmazás).

```

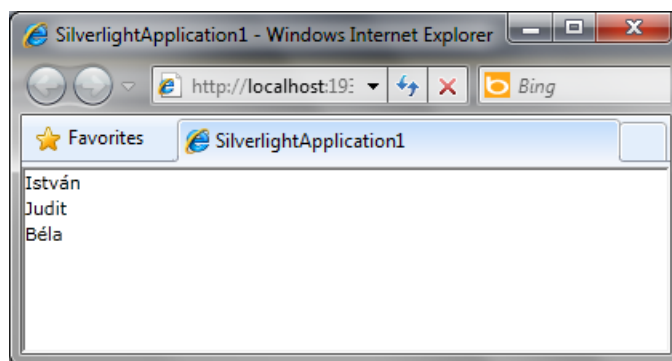
namespace SilverlightApplication1
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }
}

```

```
public partial class MainPage : UserControl
{
    public MainPage()
    {
        InitializeComponent();
        List<Person> nameList = new List<Person>()
        {
            new Person() { Name = "István", Age = 24 },
            new Person() { Name = "Judit", Age = 24 },
            new Person() { Name = "Béla", Age = 22 },
        };
        ic.ItemsSource = nameList;
        ic.DisplayMemberPath = "Name";
    }
}
```



3-14 ábra: *ItemsControl a DisplayMemberPath beállítása nélkül*



3-15 ábra: *a DisplayMemberPath beállítva*

ListBox

A **ListBox** az **ItemsControl**-ből eredő **Selector** (absztrakt) osztályból származik, amely lehetővé teszi, hogy elemeket válasszunk ki a megjelenített listából. A **ListBox** minden elemének egy-egy **ListBoxItem** objektum feleltethető meg, amely a **ContentControl** osztályból származik.

A **ListBox** elemeit a szokásos módon az **ItemsSource** tulajdonságon keresztül adhatjuk meg, de választhatjuk azt is, hogy rögtön a definícióban feltöltjük, ahogyan ezt a következő kódrészlet is mutatja:

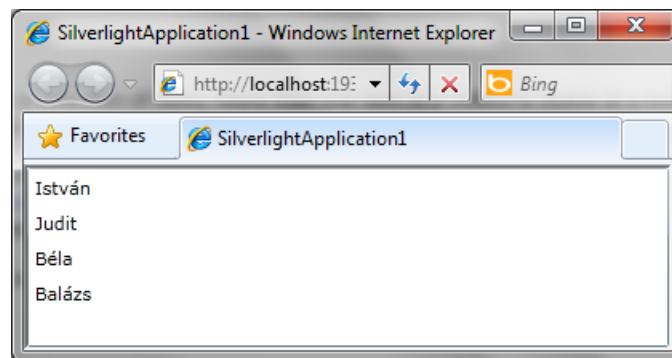
```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk">
```

```

mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
<Grid>
    <ListBox x:Name="lbox">
        <ListBox.Items>
            <ListBoxItem Content="István" />
            <ListBoxItem Content="Judit" />
            <ListBoxItem Content="Béla" />
            <ListBoxItem Content="Balázs" />
        </ListBox.Items>
    </ListBox>
</Grid>
</UserControl>

```

Az alábbi kép (3-16 ábra) mutatja az eredményt:



3-16 ábra: A `ListBox` vezérlő

Az elemek kiválasztásának módját a **SelectionMode** tulajdonsággal állíthatjuk be, alapértelmezett módon egy időben csak egy elemet (**Single** érték) jelölhetünk ki. A **Multiple** és **Extended** értékek megengedik több elem kiválasztását, de utóbbi esetén nyomva kell tartani a **Ctrl** gombot.

A **SelectionChanged** eseménnyel kezelhetjük a kiválasztott elemek listáját, ennek az eseménynek az eseménykezelője **SelectionChangedEventArgs** típusú paramétert kap, amelynek **AddedItems** illetve **RemovedItems** tulajdonságai tömbben tárolják az aktuálisan illetve az előzőleg kiválasztott elemeket.

A **ListBox** objektumon közvetlenül hívható **SelectedItem**, **SelectedItems** és **SelectedValue** tulajdonságok ugyanezt az információt adják vissza.

Mindkét módszer **object** típusú elemeket ad vissza, ezeket először **ListBoxItem** típusra kell konvertálni, hogy hozzáférjünk az értékekhez (a **ListBox Content** tulajdonságával). A következő forráskód erre mutat példát:

```

private void lbox_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    string s = string.Format("A kiválasztott elem: {0}",
        (e.AddedItems[0] as ListBoxItem).Content);
    MessageBox.Show(s);
}

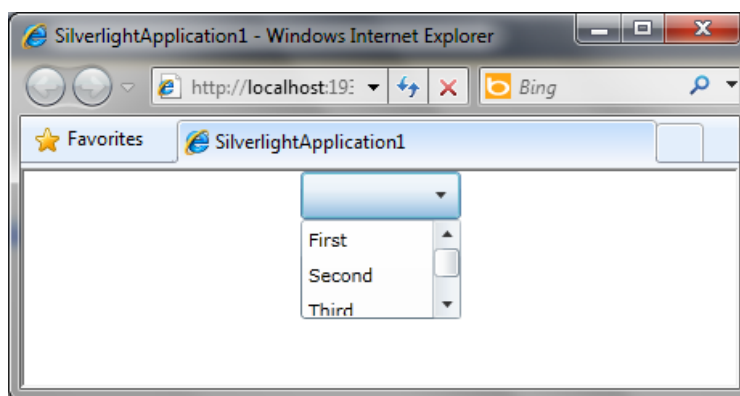
```

ComboBox

A **ComboBox** vezérlő tulajdonképpen két részből áll, egy **TextBox**-ből és egy lenyíló **ListBox**-ből. Ez utóbbi miatt használata gyakorlatilag teljesen megegyezik a **ListBox**-éval, azt leszámítva, hogy csak egyetlen elemet tudunk kiválasztani, illetve az elemeket ezúttal **ComboBoxItem** objektumok jelenítik meg.

A következő egyszerű példa futás közbeni eredménye a 3-17 ábrán látszik.

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid>
        <ComboBox x:Name="cbox" VerticalAlignment="Top" Width="100" Height="30">
            <ComboBox.Items>
                <ComboBoxItem Content="First" />
                <ComboBoxItem Content="Second" />
                <ComboBoxItem Content="Third" />
                <ComboBoxItem Content="Fourth" />
            </ComboBox.Items>
        </ComboBox>
    </Grid>
</UserControl>
```



3-17 ábra: A ComboBox vezérlő

A **ComboBox** eseményei és tulajdonságai megegyeznek a **ListBox** jellemzőivel.

TreeView

A **TreeView** vezérlő fa-struktúrában képes megjeleníteni elemeket. Minden eleme **TreeViewItem** típusú, amely maga is **ItemsControl** leszármazott, vagyis tartalmazhat több elemet, beleértve további **TreeViewItem** objektumokat is. A **TreeViewItem** osztály a **HeaderedItemsControl** osztályból származik, ezért a tartalmazott listán kívül egy fejléccel (**Header**) is rendelkezik, amely az adott csomópont felirata lesz.

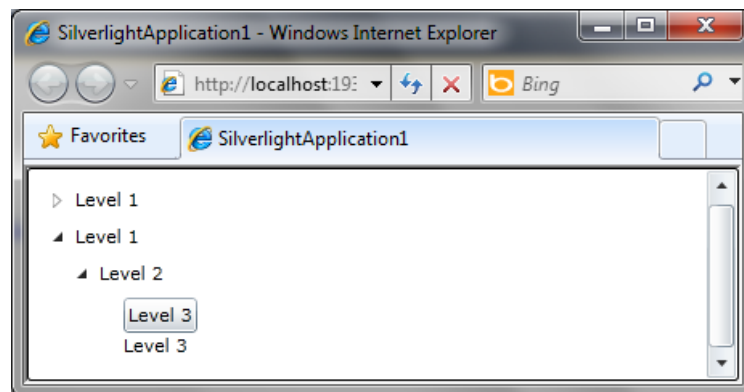
Az alábbi definíció egy többszintű **TreeView** objektumot ír le, eredménye a 3-18 ábrán látható:

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid>
        <sdk:TreeView x:Name="trvw">
            <sdk:TreeView.Items>
                <sdk:TreeViewItem Header="Level 1">
                    <sdk:TreeViewItem.Items>
                        <sdk:TreeViewItem Header="Level 2" />
                    </sdk:TreeViewItem.Items>
                </sdk:TreeViewItem>
            </sdk:TreeView.Items>
        </sdk:TreeView>
    </Grid>
</UserControl>
```

```

        <sdk:TreeViewItem Header="Level 1">
            <sdk:TreeViewItem.Items>
                <sdk:TreeViewItem Header="Level 2">
                    <StackPanel>
                        <Button Content="Level 3" />
                        <TextBlock Text="Level 3" />
                    </StackPanel>
                </sdk:TreeViewItem>
            </sdk:TreeViewItem.Items>
        </sdk:TreeViewItem>
    </sdk:TreeView.Items>
</sdk:TreeView>
</Grid>
</UserControl>

```



3-18 ábra: A TreeView vezérlő

A **TreeView** működése hasonló a **ListBox**-éhoz, a **SelectedItemChanged** esemény mutatja a kiválasztott elem változását, amelyet a **SelectedItem** tulajdonság segítségével kérhetünk le.

TabControl

A **TabControl** vezérlő segítségével az alkalmazás ugyanazon területét oszthatjuk szét több elem között. Ez a vezérlő is saját típust használ gyermekei leírására (**TabItem**), amely a **HeaderedItemsControl** osztályból származik, és a **Header** tulajdonság értéke jelenik meg az egyes lapok tetején.

A **TabControl** kiválóan alkalmas a Silverlightban nem támogatott MDI (*Multiple Document Interface*) megjelenítési minta kiváltására.

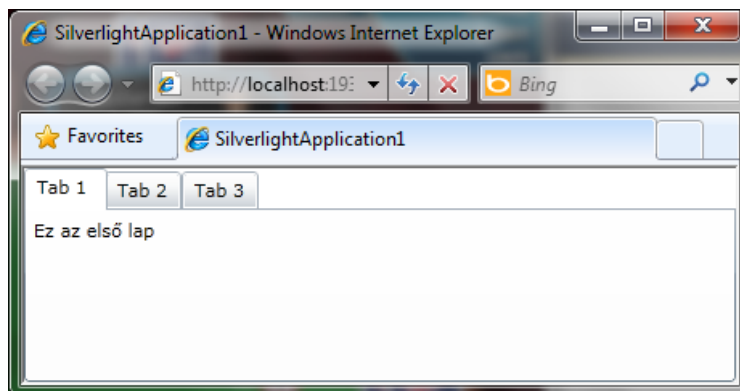
Következzen egy példa, az eredményt a 3-19 ábra mutatja.

```

<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid>
        <sdk:TabControl>
            <sdk:TabControl.Items>
                <sdk:TabItem Header="Tab 1">
                    <TextBlock Text="Ez az első lap" />
                </sdk:TabItem>
                <sdk:TabItem Header="Tab 2">
                    <TextBlock Text="Ez a második lap" />
                </sdk:TabItem>
                <sdk:TabItem Header="Tab 3">
                    <TextBlock Text="Ez a harmadik lap" />
                </sdk:TabItem>
            </sdk:TabControl.Items>
        </sdk:TabControl>
    </Grid>
</UserControl>

```

```
</sdk:TabControl.Items>
</sdk:TabControl>
</Grid>
</UserControl>
```



3-19 ábra: A *TabControl* vezérlő

A **TabControl** eseményeinek és tulajdonságainak használata megegyezik a többi **ItemsControl** leszármazottával.

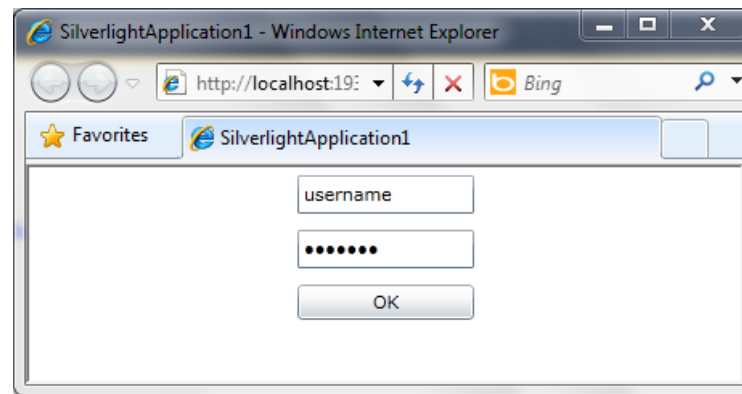
Egyéb vezérlők

Ebben a szakaszban olyan vezérlőket fogunk megvizsgálni, amelyek nem férnek be egyértelműen az eddig tárgyalt kategóriákba, viszont nagyon gyakran használjuk őket.

TextBox

A legalapvetőbb adatbeviteli vezérlő. A következő példában egy variánsával, a **PasswordBox**-szal együtt szerepel, ez utóbbi a beírt karakterek helyett a **PasswordChar** tulajdonságában megadott karaktert jeleníti meg. A kód eredményét a 3-20 ábra mutatja be.

```
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
  mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
  <StackPanel Width="120">
    <TextBox Margin="5" />
    <PasswordBox Margin="5" />
    <Button Content="OK" Margin="5" />
  </StackPanel>
</UserControl>
```



3-20 ábra: TextBox és PasswordBox vezérlők

A **TextBox** tartalmát a **Text**, míg a **PasswordBox**-ét a **Password** tulajdonsággal kérdezhetjük le.

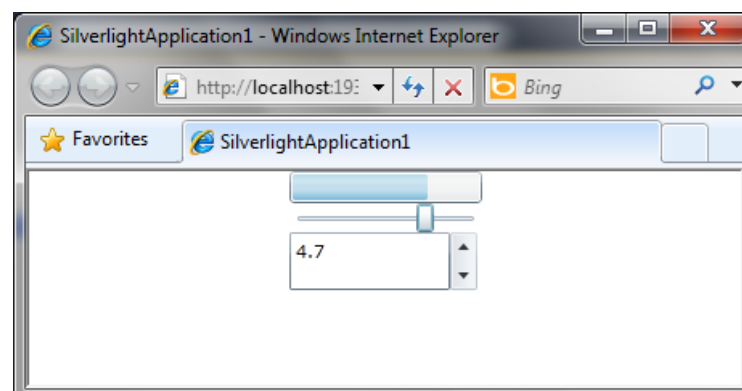
ScrollBar, Slider és ProgressBar

A három vezérlő közül az első kettő hasonló feladatot lát el, vagyis egy adott intervallumon belül beállíthatunk egy tetszőleges értéket. A **ProgressBar** egy folyamat állapotát mutatja, így némileg kakukktojásnak tűnhet, viszont a három vezérlő logikai és vizuális fája, illetve tulajdonságai mégis nagyon hasonlóak, közel egyformák.

A következő példa adatkötés használatával mutatja be ezeket az elemeket, az eredmény a 3-21 ábrán látható.

```
<UserControl x:Class="SilverlightApplication1.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
  mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
  <StackPanel Width="120">
    <ProgressBar Minimum="0" Maximum="100"
      Height="20" Value="{Binding ElementName=s11, Path=Value}" />
    <Slider x:Name="s11" Minimum="0" Maximum="100" />

    <StackPanel Orientation="Horizontal">
      <TextBox Width="100" Text="{Binding ElementName=sb1, Path=Value}" />
      <ScrollBar x:Name="sb1" Minimum="0" Maximum="100" />
    </StackPanel>
  </StackPanel>
</UserControl>
```



3-21 ábra: ProgressBar, Slider és ScrollBar vezérlők

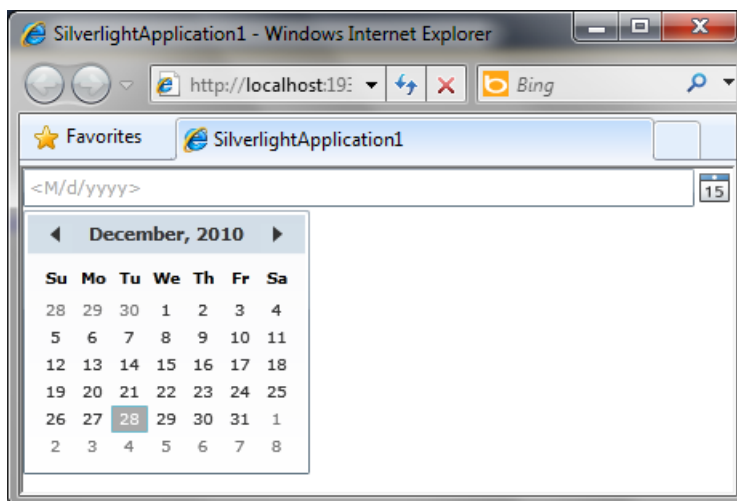
A **Minimum** és **Maximum** tulajdonságokkal a felvehető értékek határait állapítjuk meg, míg a **Value** tulajdonság az aktuális értéket adja vissza.

Calendar és DatePicker

Ezek a vezérlők lehetővé teszik dátumok kiválasztását. A **Calendar** egy standard naptárat jelenít meg, míg a **DatePicker** egy **TextBox** és **Calendar** vezérlő keveréke. A **DatePicker** használatával egyrészt kézzel beírhatjuk a dátumot, másrészt a grafikus kiválasztást támogató **Calendar** eltüntethető, így nem foglal helyet.

A következő definícióban a felülethez adunk egy **DatePicker** vezérlőt, azt a 3-22 ábrán láthatjuk futás közben.

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <StackPanel>
        <sdk:DatePicker />
    </StackPanel>
</UserControl>
```



3-22 ábra: A DatePicker vezérlő

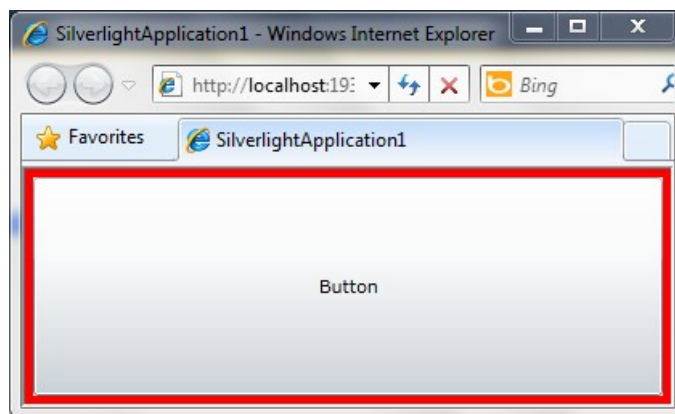
Mindkét vezérlő a **SelectedDateChanges** eseménnyel jelzi, hogy megváltozott a kiválasztott dátum.

Border

A **Border** a **TextBlock**-hoz hasonlóan a **FrameworkElement** leszármazottja, elsődlegesen a felület csinosításához használjuk. Ez a vezérlő — ahogyan a neve is sugallja — keretet képez bármely a felületen megjelenő elem körül. Hasonlóképpen működik, mint a **ContentControl** osztály, a **Child** tulajdonságában kell definiálnunk a belefoglalt elemet.

Az alábbi definícióban egy gomb köré rajzolunk keretet, a **BorderThickness** tulajdonság annak vastagságát, a **BorderBrush** a színét tárolja (az eredmény a 3-23 ábrán).

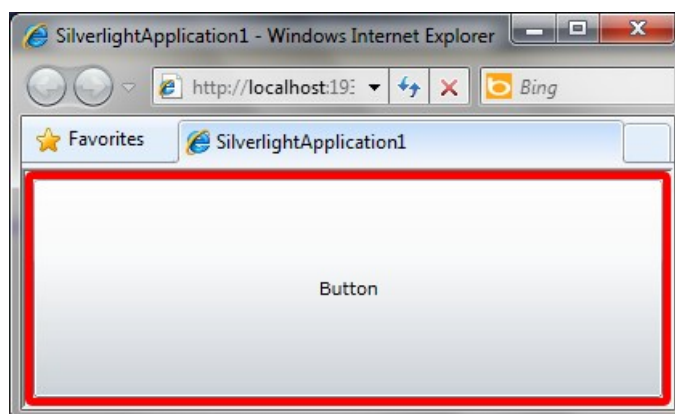

```
<UserControl x:Class="SilverlightApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
    <Grid>
        <Border BorderThickness="5" BorderBrush="Red">
            <Button Content="Button" />
        </Border>
    </Grid>
</UserControl>
```



3-23 ábra: Gomb köré rajzolt Border

A **CornerRadius** tulajdonság segítségével lekerekíthetjük a keret sarkait:

```
<Border BorderThickness="5" BorderBrush="Red" CornerRadius="2">
```



3-24 ábra: Lekerekített sarkok

Összefoglalás

Ebben a fejezetben megvizsgáltuk a Silverlight vezérlőinek osztályhierarchiáját, betekintést nyerhettünk a vezérlők működésének alapjaiba. A „lookless” vezérlők igazán könnyen módosíthatóak, tetszés szerinti megjelenéssel ruházhatjuk fel őket, a korábbi technológiáktól eltérően nem szükséges hosszú kódsorokat leírunk. A fejezet második felében láthattuk, hogyan működnek a vezérlők a gyakorlatban, megismerkedtünk a főbb altípusokkal és ezek jellegzetességeivel.

4. fejezet: Animáció és Média

Ebben a fejezetben megismerkedünk a Silverlight alapvető grafikai, animációs és média képességeivel. Ezen esszenciális ismeretek nélkül nem lennénk képesek a technológia nyújtotta rugalmasságot és modern alkalmazásfelületépítési képességeket kihasználni úgy, hogy megrendelőink egy minden elvárást teljesítő terméket kapjanak kézhez.

A példák során mindenhol megmutatom, hogyan tudunk kódból és ahol lehet az Expression Blend használatával is elkészíteni egy adott elemet. Teljes projekteket nem fogunk készíteni, helyette apró darabokat mutatok be, amelyek egy üres projektsablont használva vagy egy meglévő alkalmazásba beágyazva használhatók.

Alakzatok rajzolása a Silverlightban

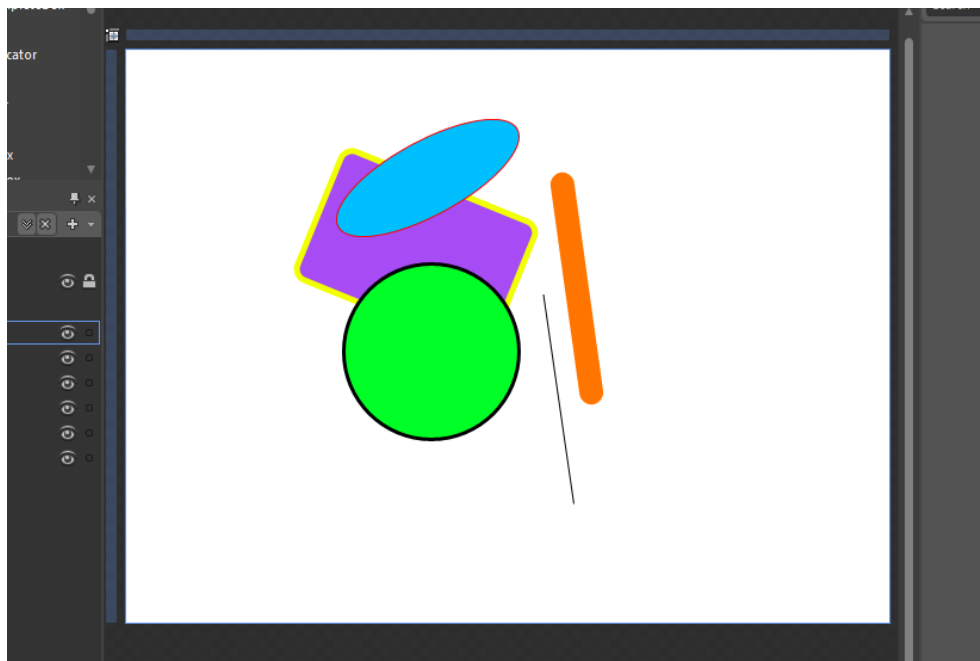
A .NET 3.0-ban megjelent *Windows Presentation Foundation* (WPF) egyik legfontosabb innovációja az előző megjelenítési technológiákhoz képest az, hogy vektorgrafikus alapon képes felhasználó felületeinket kezelni és megjeleníteni. Ez hatalmas rugalmasságot ad mind a felhasználói élmény szempontjából, mind pedig az alkalmazás elkészítése során. Nyugodtan változhat a futtatási környezet, alkalmazásunk mindenhol ugyanúgy fog kinézni, nem lesznek többé szétesett, pixelesse vált felhasználói felületek. A Silverlight ezt a képességet megörökölte nagytestvérétől, és számos beépített eszközt tartalmaz, amely segít ebben a világban otthonosan mozognunk és szemet kápráztató alkalmazásokat készítenünk. Lehetőségünk van alapvető geometriai alakzatokkal és azok kombinációival dolgozni, de saját elképzeléseinket is könnyedén megvalósíthatjuk. Ha pedig már kész a felületünk kinézete, akkor kedvünkre animálhatjuk vagy bővíthetjük különböző médiaelemekkel azt.

Nagyon fontos megemlíteni, hogy a Silverlight implicit módon képes az alakzatok számítását és megjelenítését elvégezni, nem nekünk kell saját kezűleg leprogramoznunk, mint pl. a számítógépes játékok esetében.

A számítógépes játékok nem eseményvezérelt alapon működnek, hanem egyetlen óriási végtelen ciklus foglalja magába a teljes alkalmazást, amely minden egyes iterációban végrehajtja a benne található kódot. A ki- és bemenet ellenőrzése minden egyes iterációban megtörténik, és csak ezután kerül megjelenítésre a grafikus tér, másodpercenként körülbelül 60-szor.

Egyszerűbb geometriai formák rajzolása

Először nézzük meg, hogy milyen alapvető geometriai alakzatokat tudunk használni! Csupán néhány elemünk van: a négyszög (*rectangle*), az ellipszis (*ellipse*), a vonal (*line*) és a sokszög (*polygon*). A Silverlight a megjeleníteni kívánt elemeket teljesen homogén módon kezeli, ez azt jelenti, hogy az alakzatok ugyanazokkal az alaptulajdonságokkal és eseményekkel rendelkeznek, mint bármelyik másik vezérlő, ezeket a 4-1 táblázat foglalja össze. Ugyanúgy tudjuk őket elrendezni a felületen, és ugyanúgy képesek pl. a billentyűzet vagy az egér eseményeit elkapni. Erre mutat példát a 4-1 ábra.



4-1 ábra: Alakzatok Expression Blendben

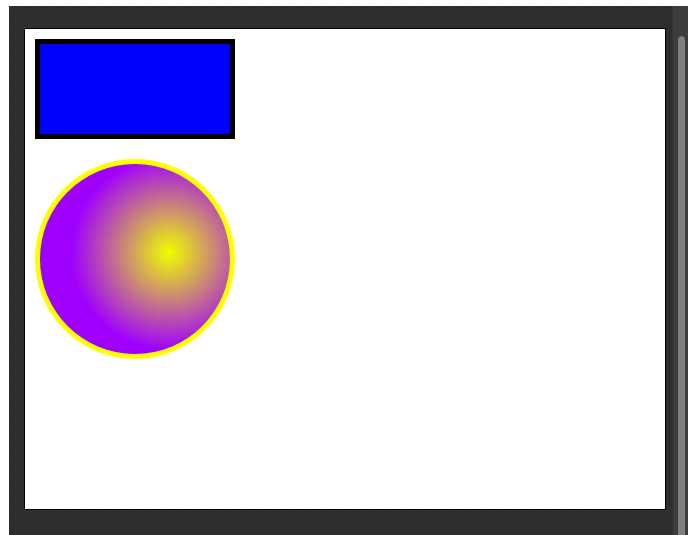
4-1 táblázat: A geometriai alakzatokat leíró objektumok alaptulajdonságai

Tulajdonság neve	Rövid leírás
Fill	Az adott vezérlő kitöltéséért felelős tulajdonság. Különböző módokat használhatunk a színezéshez, ezeket a fejezet későbbi részeiben tárgyaljuk.
Stroke	Az adott alakzat szegélyének a színét írja le. Ennek a segítségével alakzatainkat kiemelhetjük a többi közül.
StrokeThickness	A keret vastagságát írja le lebegőpontos számokkal.
Width, Height	Az adott alakzat szélessége és magassága pixelekből mérve.

A **Rectangle** és az **Ellipse** vezérlők használatára nézzünk meg egy egyszerű példát! Az alábbi kód rajzol egy téglalapot és egy kört (azok kitöltéséről még beszélni fogok).

```
<StackPanel x:Name="LayoutRoot" Background="White">
  <Rectangle Fill="Blue" Height="100" Stroke="Black"
    Width="200" HorizontalAlignment="Left"
    Margin="10" StrokeThickness="5"/>
  <Ellipse Height="200" Stroke="#FFDFF0"
    StrokeThickness="5" Width="200"
    HorizontalAlignment="Left" Margin="10">
    <Ellipse.Fill>
      <RadialGradientBrush Center="0.675,0.468"
        GradientOrigin="0.675,0.468">
        <GradientStop Color="#FFF1FF00"/>
        <GradientStop Color="#FFA000FF" Offset="1"/>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
</StackPanel>
```

Ez a kódrészlet a 4-2 ábrán látható módon rajzolja ki az alakzatokat:

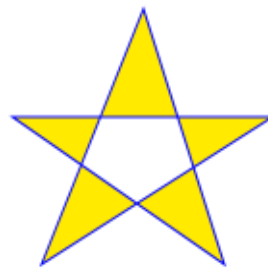


4-2 ábra: A Rectangle és Ellipse vezérlők

Sokszögeket alkotni már sokkal izgalmasabb feladat. A 4-1 táblázatban leírtak mellett két fontos új tulajdonságot vezet be a **Polygon** osztály. Egyike a **Points** tulajdonság, amely a sokszög csúcsait tartalmazza vesszővel elválasztott párokként felsorolva és a párokat szóközzel elválasztva. Továbbá rendelkezik egy **FillRule** tulajdonsággal is, amely a sokszög kitöltésének módját írja le.

```
<Grid x:Name="LayoutRoot" Background="White">
  <Polygon Points="15,200 68,70 110,200 0,125 135,125"
    Fill="#FFFEA00" Margin="0" FillRule="Nonzero"
    Stroke="#FF0004FF" HorizontalAlignment="Center"
    VerticalAlignment="Center" />
</Grid>
```

A kód a 4-3 ábrán megjelenő csillag alakzatot eredményezi.



4-3 ábra: A Polygon vezérlővel leírt alakzat

Egyszerű vonalakkal dolgozni sem nehezebb. Itt a két végpontot kell leírnunk az **X1**, **Y1**, **X2**, **Y2** tulajdonságokkal, a vonal vastagságát pedig a **StrokeThickness** adja meg. Amit azonban érdekesebb lehet beállítani, az a kezdő- és végpont rajzolása. Beállíthatunk speciális alakzatokat, amelyek a következők:

- **Triangle** (háromszög)
- **Square** (négyzet)
- **Round** (lekerekített)
- **Flat** (nincs speciális végződés)

A **Line** vezérlőelem használatát mutatja be az alábbi példa, amelynek eredményét a 4-4 ábra illusztrálja:

```
<Grid x:Name="LayoutRoot" Background="White">
  <Line X1="84" Y1="234" X2="587" Y2="88" Fill="#FF08FF00"
    Height="147" Stretch="Fill"
    Stroke="Red" StrokeThickness="20"
    StrokeStartLineCap="Triangle"
    StrokeEndLineCap="Round"/>
</Grid>
```

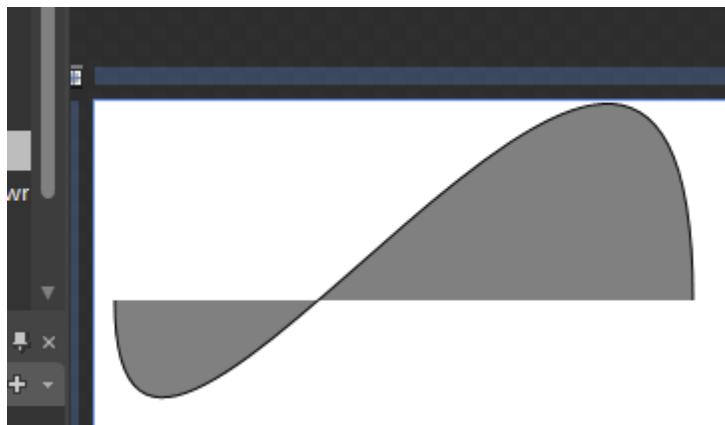


4-4 ábra: A *Line* vezérlő használata

Saját alakzatok ábrázolása

Van egy speciális vezérlő, amely kimondottan egyéni, összetett alakzatok megjelenítésére szolgál, ez a **Path**. Magáról a vezérlőről nem sok újat lehet elmondani, ugyanazokkal a tulajdonságokkal rendelkezik, mint az egyszerű alakzatok. Azonban az összetett geometriák leírására egy speciális saját „mini nyelvezetet” használ. A formák leírását a **Data** tulajdonságon keresztül adhatjuk meg.

A 4-5 ábra a **Path** vezérlő használatát mutatja be:



4-5 ábra: Az egyszerű „görbe” megrajzolása a *Path* vezérlővel

A 4-5 ábrán látható alakzat megrajzolását az alábbi egyszerű kód végzi a **Path** használatával:

```
<Canvas>
  <Path Stroke="Black" Fill="Gray"
    Data="M 10,100 C 10,300 300,-200 300,100" />
</Canvas>
```

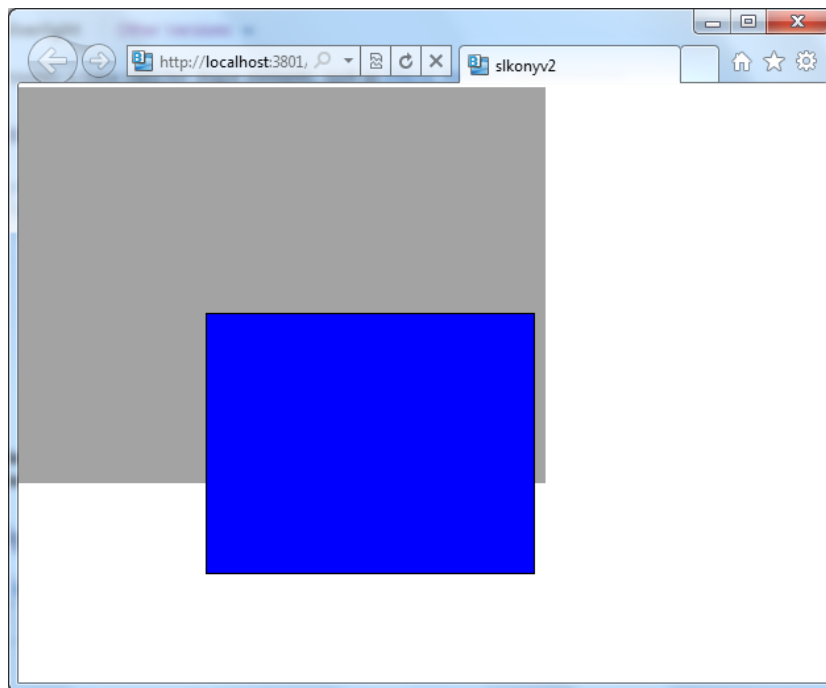
A fenti kódból is jól látható, hogy a **Data** tulajdonság értékének megadásakor valamilyen speciális leírást használtunk. Nézzük, melyek az ezekben szereplő elemek!

- **M** – kezdőpont, innen indul ki az alakzatunk
- **L** – az alakzat végpontja
- **H** – vízszintes vonal
- **V** – függőleges vonal
- **C** – köbalapú Bezier-görbe
- **Q** – négyzetes Bezier-görbe

Ezek mellett még számos más alakzatot használhatunk, amelyeknek teljes listája és szintaktikája megtalálható a [http://msdn.microsoft.com/en-us/library/cc189041\(v=vs.95\).aspx](http://msdn.microsoft.com/en-us/library/cc189041(v=vs.95).aspx) oldalon.

Vágási felületek kialakítása

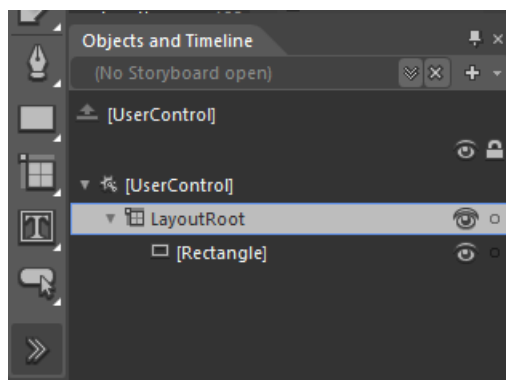
Nézzük meg a 4-6 ábrát, azon valami furcsaságot láthatunk!



4-6 ábra: Furcsa módon megjelenő téglalap

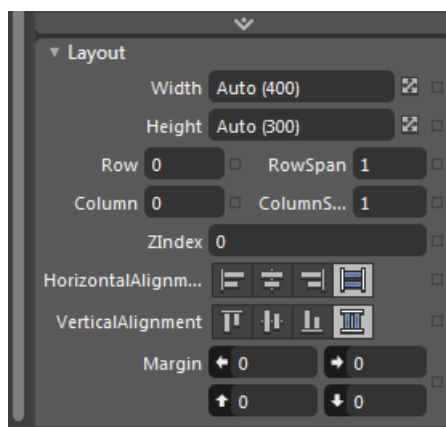
Az alkalmazás mérete 400×300 pixel, a négyszög pedig, amit rajzoltunk, csúnyán kilóg az alkalmazás területéről. Ahhoz, hogy az ilyen eseteket elkerüljük, lehetőségünk van vágási felületek létrehozására. A Silverlight vezérlők rendelkeznek egy **Clip** tulajdonsággal. Ennek egy sokszög paraméteres leírását tudjuk megadni, amelyet magunktól kiszámolni nem minden esetben egyszerű feladat (ne csak erre az egyszerű esetre gondoljunk, hanem pl. egy számológép programra, amelyben a gombok lenyomását kell szimulálnunk). A Blend használatával rendkívül könnyű egy ilyen vágási felületet kialakítani.

Válasszuk ki az adott Layout vezérlőt, amelyre a vágást alkalmazni szeretnénk (4-7 ábra)!



4-7 ábra: A LayoutRoot elem kiválasztása az Objects and Timeline panelben

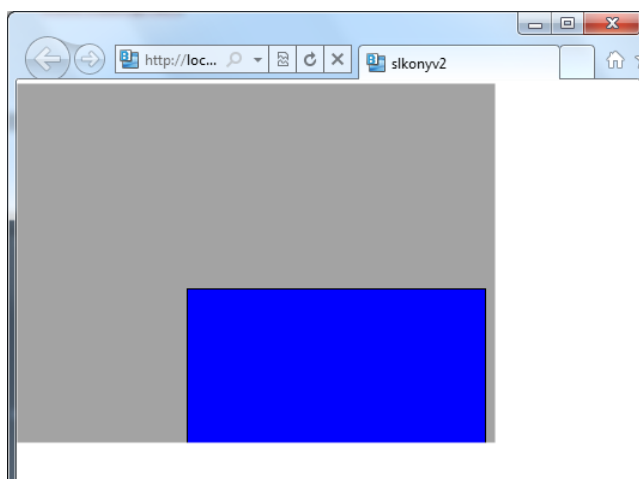
Hozunk létre egy új **Rectangle** vezérlőt (ezt legegyszerűbben a bal oldalon található menüoszlop segítségével tehetjük meg), amelynek a méreteit állítsuk pontosan akkorára, mint a Layout vezérlő (4-8 ábra)!



4-8 ábra: A Rectangle vezérlő méreteinek beállítása

Ha ezzel megvagyunk, válasszuk ki egyszerre a Layout vezérlőt és az imént létrehozott **Rectangle** vezérlőt, majd nyomjuk meg a CTRL+7 gombokat (vagy menüben válasszuk az Object ➤ Path ➤ Make Clipping Path parancsot)!

Ha most futtatjuk az alkalmazást, akkor a négyzet „kilógó” részei már nem fognak megjelenni, amint ezt a 4-9 ábra is mutatja.



4-9 ábra: A négyzet megjelenítése vágási felületek alkalmazásával

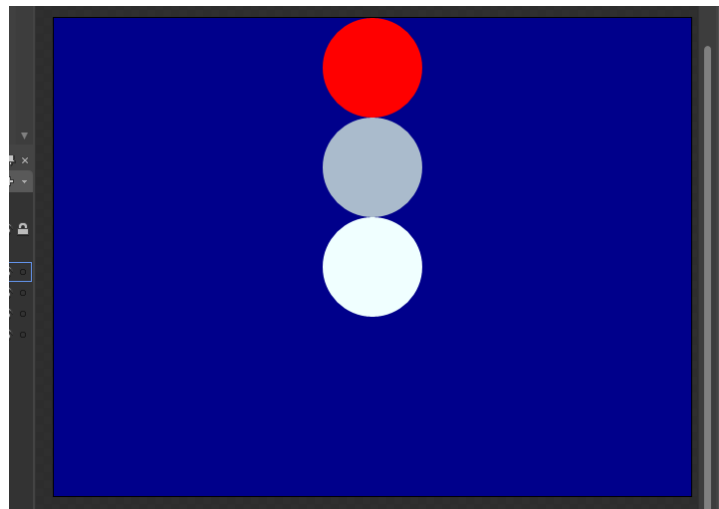
Az alakzatok kitöltése és színezése

Egyszerű kitöltés

Az egyszerű kitöltés, amint az a nevében is benne van, egyetlen adott színnel tölti ki az alakzatunkat. Ezt a kitöltést a **Fill** tulajdonságon keresztül végezhetjük el, többféle formában is.

A 4-10 ábra azt mutatja be, amit az alábbi rövid kódrészlet segítségével hozhatunk létre:

```
<StackPanel x:Name="LayoutRoot" Background="DarkBlue">
  <Ellipse Width="100" Height="100" Fill="Red" />
  <Ellipse Width="100" Height="100" Fill="#FFAABBCC" />
  <Ellipse Width="100" Height="100">
    <Ellipse.Fill>
      <SolidColorBrush Color="Azure" />
    </Ellipse.Fill>
  </Ellipse>
</StackPanel>
```



4-10 ábra: Egyszerű kitöltésű megjelenítés a Blend tervezői felületén

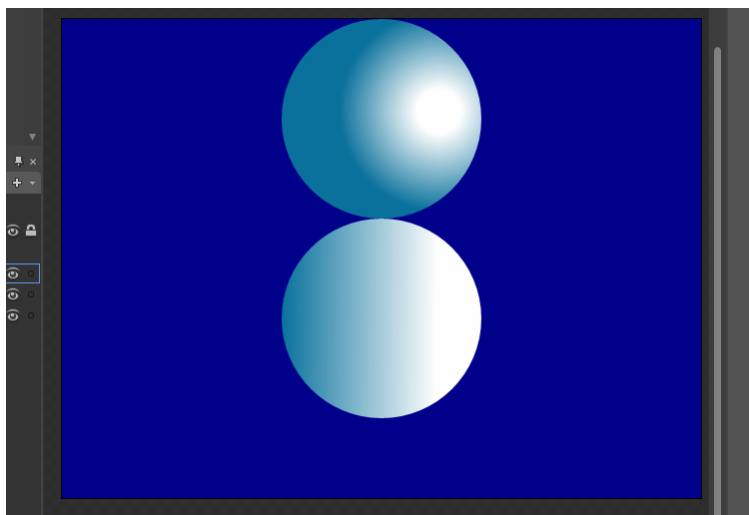
Ahogy a példában látható, megadhatjuk szövegesen az alapszínek neveivel, vagy hexadecimálisan a „#” karaktert követően az **#AARRGGBB** színkód formulára illeszkedve (ahol az egyes betűk a színcsatornákat jelölik).

Amikor alkalmazásunkat lefordítjuk, ezeket a színeket mind a **SolidColorBrush** osztály egy-egy példányára reprezentálja. A fenti példának megfelelően XAML-ben is használhatjuk ezt explicit módon, illetve kódból csak így tudunk új kitöltéseket definiálni.

Színátmenetes kitöltés

Tudunk több szín közötti átmenetet — ún. gradienset — definiálni. Ezzel a módszerrel több vezérpontot adunk meg, amelyek mind egy-egy színt írnak le, a színek között pedig lineáris interpolációval kerülnek kiszámításra az átmenetek. Fontos még megadni, hogy mely (X1, Y1) koordinátpártól mely (X2, Y2) párosig szeretnénk az átmenetet definiálni. Ezeknek a használatához a Silverlight két osztályt vezet be, a **Linear-** és a **RadialGradientBrush** osztályokat. A lineáris variáns egyenletes színátmenetek képzésére alkalmas, amíg a radiális körkörösön, belülről kifelé képes a színeket átmenetessé tenni. Erre mutat példát a 4-11 ábra, illetve az alábbi kódrészlet:

```
<StackPanel x:Name="LayoutRoot" Background="DarkBlue">
  <Ellipse Width="200" Height="200" >
    <Ellipse.Fill>
      <RadialGradientBrush Center="0.787,0.463" GradientOrigin="0.787,0.463">
        <GradientStop Color="White" Offset="0.224"/>
        <GradientStop Color="#FF09719D" Offset="1"/>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
  <Ellipse Width="200" Height="200" Margin="220,0" >
    <Ellipse.Fill>
      <LinearGradientBrush EndPoint="1,0.506"
        MappingMode="RelativeToBoundingBox"
        StartPoint="0,0.494">
        <GradientStop Color="White" Offset="0.776"/>
        <GradientStop Color="#FF09719D"/>
      </LinearGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
</StackPanel>
```

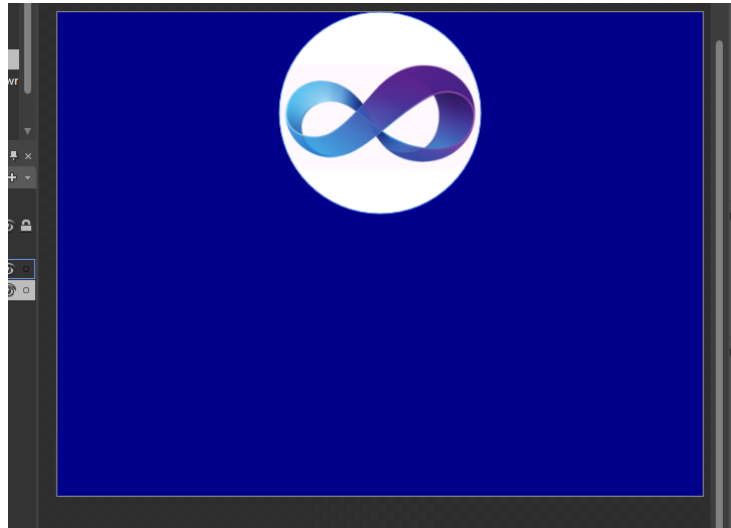


4-11 ábra: Színátmenetes kitöltések a Blend tervezői felületén

Kitöltés képpel

Most már láttuk a **SolidColorBrush** és a **GradientBrush** osztályok működését, amelyek mind előre definiált színekkel töltötték ki az adott alakzatot. Ezeken kívül lehetőségünk van képpel is kitölteni, amelyet az **ImageBrush** osztály biztosít. Csupán a kép forrását kell megadnunk (ezt a fejezet későbbi részében részletesen tárgyaljuk), és utána rögtön a kép kerül az eddigi színek helyére. Ennek a használatát a 4-12 ábra mutatja be, illetve az alábbi kódrészlet:

```
<StackPanel x:Name="LayoutRoot" Background="DarkBlue">
  <Ellipse Width="200" Height="200" >
    <Ellipse.Fill>
      <ImageBrush ImageSource="Visual-Studio-2010.png"/>
    </Ellipse.Fill>
  </Ellipse>
</StackPanel>
```



4-12 ábra: Alakzat kitöltése képminta segítségével

Kitöltés videóval

Bizony, lehetőségünk van videóval kitölteni az alakzatainkat. A videók használata nem sokkal bonyolultabb, mint a képeké, azonban szükségünk van egy plusz objektumra, amely a videó forrását képes eltárolni és a videó anyagot kezelni. Ez a **MediaElement**. Ettől az elemtől fogjuk megkapni a szükséges erőforrásokat, amellyel majd kitöltjük az adott vezérlőt. A kitöltésért a **VideoBrush** objektum a felelős.

```
<Grid x:Name="LayoutRoot" Background="White">
    <MediaElement x:Name="butterflyMediaElement"
        Source="Butterfly.wmv" IsMuted="True"
        Opacity="0.0" IsHitTestVisible="False" />

    <TextBlock Canvas.Left="5" Canvas.Top="30"
        FontFamily="Verdana" FontSize="120"
        FontWeight="Bold" TextWrapping="Wrap"
        Text="Video">

        <TextBlock.Foreground>
            <VideoBrush SourceName="butterflyMediaElement" Stretch="UniformToFill" />
        </TextBlock.Foreground>
    </TextBlock>
</Grid>
```

Képek és videók megjelenítése

A Silverlight — mint multimédiás platform — lehetőséget biztosít számunkra különböző kép-, videó- és zeneformátumok használatára, kiegészítve azokat speciális technológiákkal, mint például a *DeepZoom* vagy a *Smooth Streaming*.

Képek megjelenítése az *Image* vezérlő segítségével

XAML kód szempontjából a képek megjelenítése homogén — formátumtól független - módon történik, az **Image** vezérlőn keresztül. Azonban a kép forrásának a kijelölésére már különböző módok állnak rendelkezésünkre. Lehetőségünk van az alkalmazásunkban elhelyezni a képeket (magában a DLL modulban erőforrásként, vagy a XAP csomagban, amely magát a Silverlight alkalmazást és az erőforrásait tartalmazza), rámutatni a képek elérhetőségére a weben, vagy akár az adott kép elérését ismerve manuálisan letölteni és feldolgozni azt, majd csak a pixeladatokat átadni az **Image** vezérlőnek. Természetesen ezekkel nem merítettük ki a Silverlight képekhez kapcsolódó képességeit, mivel ezen túl

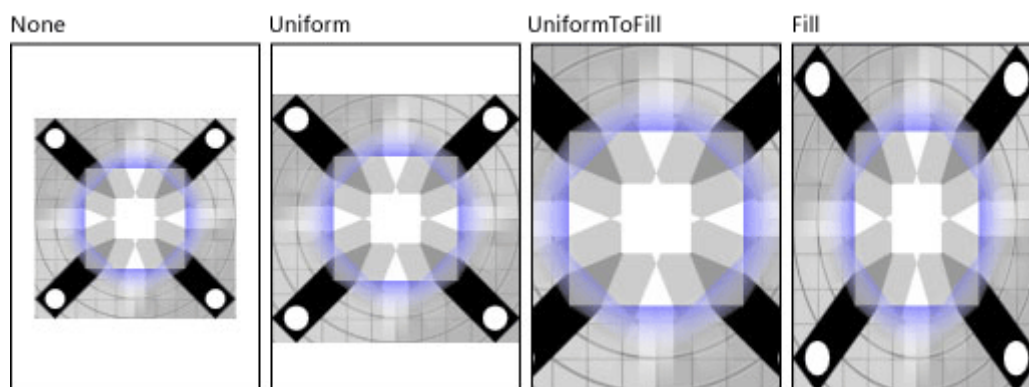
még lehetőségünk van DeepZoom (óriás felbontású képek) vagy dinamikusan készített (generált) képek megjelenítésére is.

Fontos vigyáznunk, hogy ha idegen URL-ről származik az adott erőforrás, akkor a **clientaccesspolicy.xml**-re szükség lesz a cross-domain hivatkozások miatt! Soha ne hivatkozzunk erőforrásokra, amelyek a helyi gépen helyezkednek el, mivel a Silverlight nem tud a fájlrendszerrel kommunikálni, csak ha Out-of-Browser módban fut az alkalmazás!

Az alábbi példában egy olyan képet jelenítünk meg, amely a XAP csomag része lesz fordításkor (content). A kép lehetne erőforrás (resource) is, ekkor az alkalmazás Assemblybe kerülne fordításkor.

```
<Grid x:Name="LayoutRoot" Background="DarkBlue">
  <Image Margin="0" Source="Visual-Studio-2010.png" Stretch="Fill"
    Width="200" VerticalAlignment="Center"
    HorizontalAlignment="Center" Height="200"/>
</Grid>
```

Van egy nagyon fontos beállítás képek esetében (videóknál is érvényes!), ez a méretbeli igazítás módja (Stretch), azaz ha a videó dimenziói nem az adott vezérlőhöz illeszkednek, akkor hogy töltsük ki vele a helyet, és vágjunk-e le részeket az adott képből (videóból). A 4-13 ábra bemutatja a négy különböző kitöltési módot.



4-13 ábra: Képek kitöltési módjának állítása

Videó és zene lejátszása

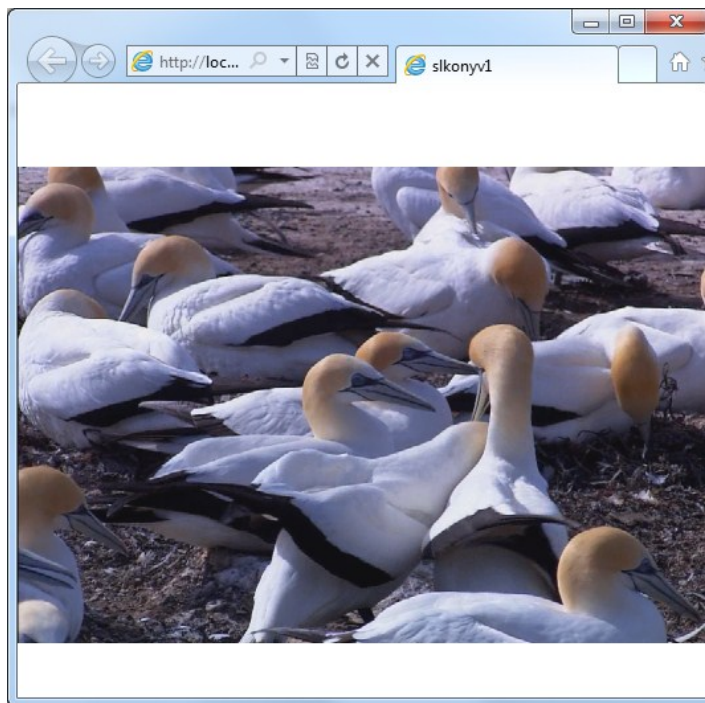
A videók és zenék használata meglehetősen hasonlít a képekére, tovább egyszerűsítve azzal a helyzettel, hogy mind a két média lejátszása ugyanazon a vezérlőn keresztül történik. Ennek a neve: **MediaElement**. A 4-2 táblázat összefoglalja a gyakrabban használt tulajdonságokat.

4-2 táblázat: A MediaElement legfontosabb tulajdonságai

Tulajdonság neve	Rövid leírás
AutoPlay	Megadhatjuk, hogy a videó-, zeneforrás betöltését követően automatikusan szeretnénk-e elkezdni a lejátszást.
Source	A médiaforrást jelöli ki. Ezen a tulajdonságon keresztül adhatjuk meg, hogy mit szeretnénk lejátszani.
Volume	[0; 1] közötti értékek segítségével megadhatjuk, hogy milyen hangerővel szeretnénk lejátszani.
IsMuted	Hangforrás némítása.
Stretch	Akárcsak a képeknél, a kitöltés módjának beállítása.

A 4-14 ábra egy minta videó lejátszását mutatja be, az alábbi forráskód segítségével:

```
<Grid x:Name="LayoutRoot" Background="White">
    <MediaElement HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Source="/Wildlife.wmv"
        Volume="0.8"/>
</Grid>
```



4-14 ábra: Videó lejátszás közben

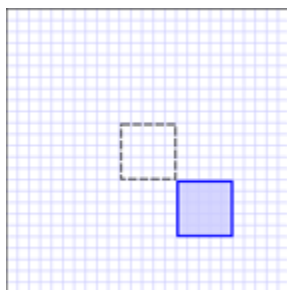
Transzformációk a felületen

A felhasználói felület mindegy egyes eleme alkalmas arra, hogy valamilyen módon transzformáljuk, vagyis a képernyőn kezdetben kiválasztott pozíciójából elmozdítsuk. Ezek a műveletek önmagukban is nagyon hasznosak, azonban a bennük rejlő lehetőségeket akkor látjuk meg, ha a felületet életre akarjuk kelteni és a transzformációkat animációk magjaként használni.

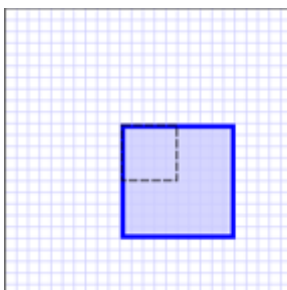
Egyszerű transzformációk

Egyszerű transzformációknak nevezzük azokat a kétdimenziós síkban végzett transzformációkat, amelyek olyan alaplátványt valósítanak meg, mint az eltolás, a nagyítás, a kicsinyítés és a forgatás. A Silverlight-ban minden vezérlő, amely a felületen elhelyezhető, örökölten tartalmaz egy **RenderTransform** tulajdonságot. Ezzel egyetlen transzformációt írhatunk le az alábbiak közül:

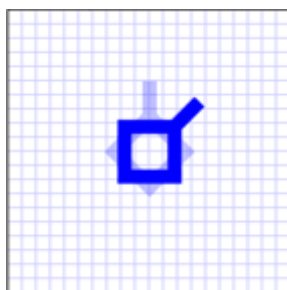
- **TranslateTransform** (eltolás, 4-15 ábra)
- **ScaleTransform** (kicsinyítés, nagyítás, 4-16 ábra)
- **RotateTransform** (forgatás, 4-17 ábra)
- **SkewTransform** (billentés, 4-18 ábra)



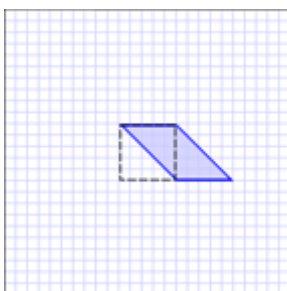
4-15 ábra: *TranslateTransform* (eltolás)



4-16 ábra: *ScaleTransform* (kicsinyítés, nagyítás)

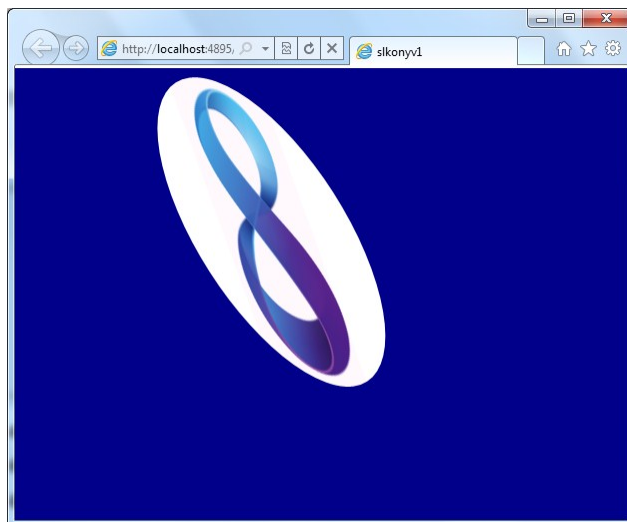


4-17 ábra: *RotateTransform* (forgatás)



4-18 ábra: *SkewTransform* (billentés)

Ha egyszerre több transzformációt is szeretnénk használni, akkor egy **TransformGroup** objektumot kell létrehoznunk, és ezen belül érvényesíteni az előbb említett elemi transzformációkat. A 4-19 ábra egy összetett transzformációt mutat be.



4-19 ábra: Egy összetett transzformáció a Silverlight-ban

Ezt a transzformációt az alábbi XAML-kód valósítja meg:

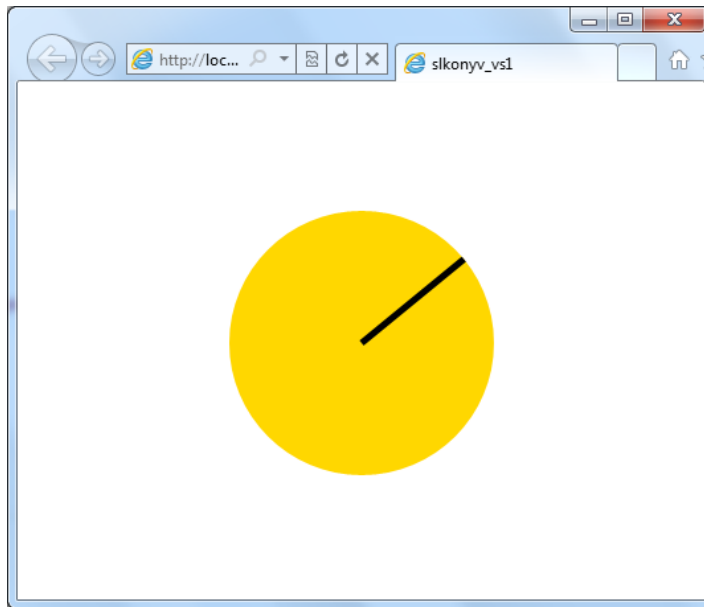
```
<StackPanel x:Name="LayoutRoot" Background="DarkBlue">
  <Ellipse Width="200" Height="200" RenderTransformOrigin="0.5,0.5" >
    <Ellipse.RenderTransform>
      <TransformGroup>
        <ScaleTransform ScaleX="1.5" ScaleY="1" />
        <TranslateTransform X="100" Y="100" />
        <RotateTransform Angle="90" />
        <SkewTransform AngleX="20" AngleY="20" />
      </TransformGroup>
    </Ellipse.RenderTransform>
    <Ellipse.Fill>
      <ImageBrush ImageSource="Visual-Studio-2010.png"/>
    </Ellipse.Fill>
  </Ellipse>
</StackPanel>
```

Jól látható a példából, hogy az egyes transzformációkhoz milyen egyedi tulajdonságok tartoznak. Az eltolás során egy X és Y párt kell megadnunk (eltolás vektor), amellyel az adott vezérlőt mozdítjuk el. Skálázáskor a **ScaleX** és **ScaleY** tulajdonságokat kell megadnunk, amelyek megmutatják, hogy adott tengelyen milyen arányban kicsinyítjük vagy nagyítjuk a vezérlőt. Forgatáskor pedig nincs más dolgunk, mint megadni egy szöveget (fokokban), amellyel a virtuális Z-tengely körül forgatjuk el az alakzatot a síkban. Fontos, hogy minden transzformációnak van egy ún. viszonyítási pontja (**RenderTransformOrigin**). Ezeket a skaláris értékeket [0, 1] intervallumon tudjuk beállítani, amely azt jelenti, hogy adott tengelyen nézve a 0 az alakzat eleje, az 1 pedig a vége, a kettő között lineárisan mutathatjuk meg a nekünk szükséges pontot. A mintakódban használt (0.5, 0.5) koordinátpár az alakzat középpontját jelöli ki.

Saját transzformációk mátrixokkal

Vannak olyan esetek, amelyeket nagyon nehéz lenne elemi transzformációkkal elvégezni, azokat megfelelően összehangolni (főleg a következő alfejezetben tárgyalt 3D transzformációk körében). Ezért a Silverlight tervezői lehetőséget biztosítottak számunkra, hogy az adott vezérlő teljes pozicionálását a kezünkbe vegyük és saját, matematikai alapú leírást adjunk a transzformációra mátrixok segítségével.

A 4-20 ábra egy másodpercmutatót transzformál a felületen mátrixok segítségével.



4-20 ábra: Másodpercmutató mátrix transzformációk segítségével

A transzformációt az alábbi kód valósítja meg:

```
<!-- XAML kód -->

<Grid x:Name="LayoutRoot" Background="White" Width="200" Height="200">
    <Ellipse Width="200" Height="200" Fill="Gold" />
    <Line x:Name="myLine" X1="0" X2="0" Y1="0" Y2="-100"
        Fill="Black" Stroke="Black" StrokeThickness="5"
        Margin="100,100,0,0"/>
</Grid>

// --- C# kód

DispatcherTimer timer;
int count;

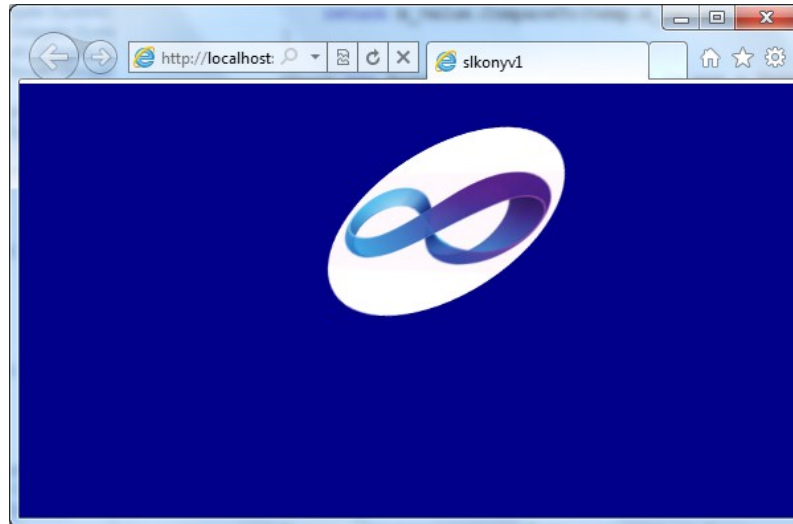
public MainPage()
{
    InitializeComponent();
    this.Loaded += new RoutedEventHandler(MainPage_Loaded);
}

void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    timer = new DispatcherTimer();
    timer.Interval = TimeSpan.FromSeconds( 1 );
    timer.Tick += ( s, e1 ) =>
    {
        double angle = count++ * ( 1.0 / ( 360.0 / 60 ) );
        Matrix m = new Matrix( Math.Cos( -angle ), -Math.Sin( -angle ),
                               Math.Sin( -angle ), Math.Cos( -angle ),
                               0, 0 );
        myLine.RenderTransform = new MatrixTransform() { Matrix = m };
    };

    timer.Start();
}
```


A Silverlight 3D képességei

A Silverlight 3-ban jelent meg a 3 dimenziós transzformációk támogatása. Sajnos a WPF-fel ellentétben még nem valódi 3D kiterjesztésről beszélünk, csupán ügyesen használható „szemfényvesztésről”. Elsőre ez a szó dehonesztálón csenghet a Silverlighttal kapcsolatban. Azonban ez korántsem így van, ez csupán azt jelenti, hogy nem valódi 3D koordináta-rendszerben számol a Framework, hanem síkra levetített transzformációkat alkalmaz, ezeket nevezzük perspektivikus transzformációknak. Minden egyes **UIElement** osztályból származó vezérlő örököl egy **PlaneProjection** nevezetű tulajdonságot, amelyen keresztül a forgatás alapú transzformáció elvégezhető (itt is van lehetőségünk mátrixok használatára, azonban erre a könyv keretein belül külön nem térünk ki). A 4-21 ábra és a hozzá tartozó kódrészlet erre mutat példát.



4-21 ábra: Térbeli transzformáció Silverlight segítségével

A 4-21 ábrán lévő transzformációt az alábbi XAML-kód valósítja meg:

```
<StackPanel x:Name="LayoutRoot" Background="DarkBlue">
  <Ellipse Width="200" Height="200" RenderTransformOrigin="0.5,0.5" >
    <Ellipse.Projection>
      <PlaneProjection RotationY="40" RotationX="-45"/>
    </Ellipse.Projection>
    <Ellipse.Fill>
      <ImageBrush ImageSource="Visual-Studio-2010.png"/>
    </Ellipse.Fill>
  </Ellipse>
</StackPanel>
```

Animációk a Silverlight-ban

Animációk alkalmazásával, sokkal szórakoztatóbbá és látványosabbá tehetjük a felületet, életre kelthetjük azt. A Silverlight segítségével számos lehetőségünk van színek animálására, különböző időzített mozgások elkészítésére, vagy akár oldalak közötti átfedések kialakítására. Minden animáció azonos elv alapján működik. Ahhoz, hogy animációt tudjunk létrehozni, először is szükségünk lesz egy **Storyboard** objektumra, amely definiálja, hogy a létrehozandó animációkat mely objektumra szeretnénk érvényesíteni, illetve azon belül is melyik tulajdonságra. Megszabja az animáció viselkedését is — milyen hosszan játszunk le, ismétljük-e meg, ha véget ért, stb. —, és még sok más dologban segít nekünk. Mielőtt belemennénk a részletekbe, nézzünk meg egy példát a **Storyboard** használatára!

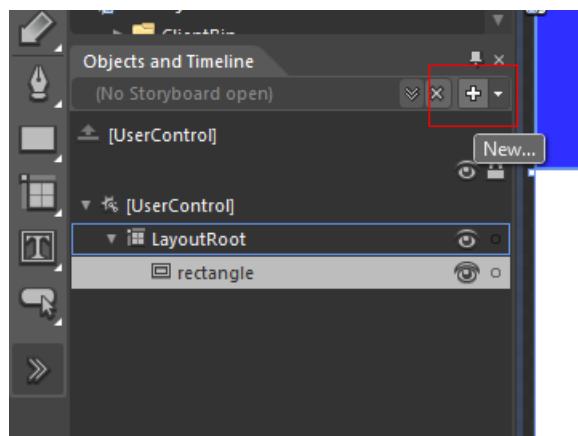
```
<Storyboard>
  <DoubleAnimation From="1.0" To="0.0"
    Duration="0:0:1"
    AutoReverse="True"
    RepeatBehavior="Forever" />
</Storyboard>
```

Ez a példa egy olyan **Storyboard** objektumot ír le, amely egy **double** típusú tulajdonsághoz kapcsolódik, és annak értékét egy másodperc alatt az 1.0 értékről a 0.0 értékre változtatja (animálja). Az animáció végeztével azt „visszafelé” megismétli, és ezt a tevékenységet megállás nélkül a végtelenségig folytatja.

Egyszerű animációk létrehozása

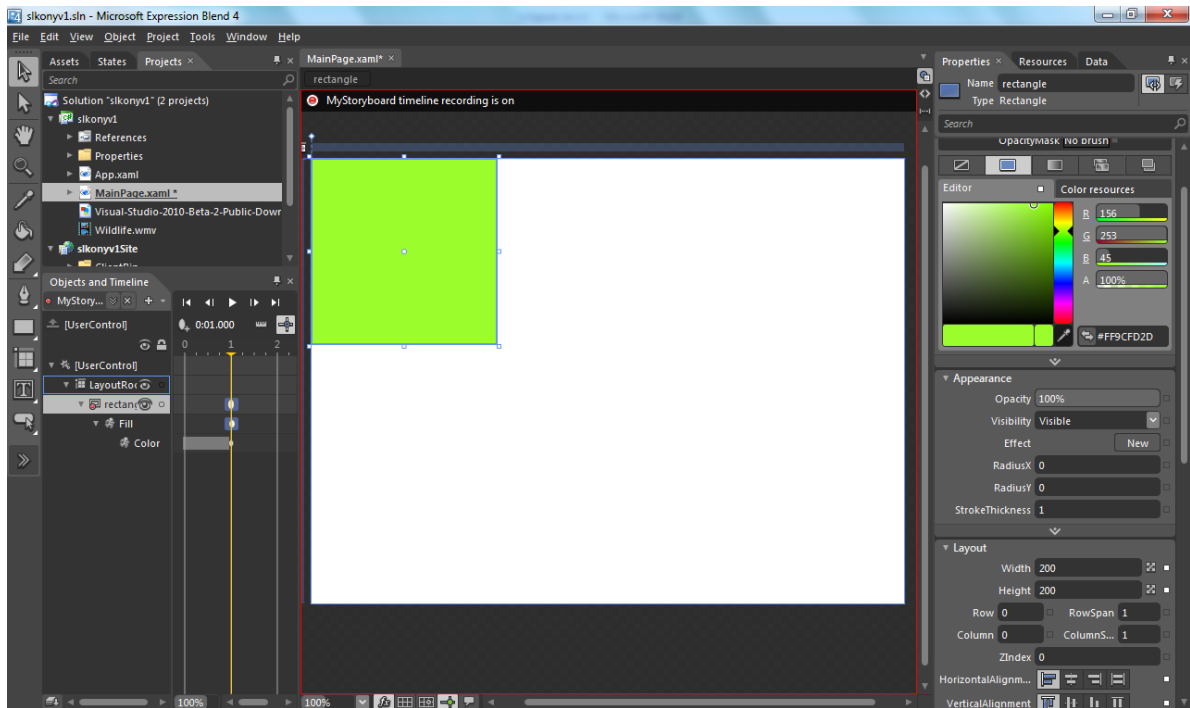
Az Expression Blend egyik legnagyobb erőssége az animációk készítésében rejlik. Az összes képességének és funkciójának az áttekintése bőven meghaladná ennek a könyvnek a terjedelmét, ezért itt csak a lényegre fókuszálok. Ahogy azt a bevezetőben említettem, az animációkat egy-egy **Storyboard** objektumon belül fogom definiálni, és a vezérlőket erőforrásként ágyazom be az alkalmazásokba. A Blendben egy animációt a legkönnyebben úgy tudunk létrehozni, hogy az Object and Timeline panelen a New gombot választjuk, ahogyan az a 4-22 ábrán is látható. Miután ez megtörtént, egy nevet adunk a **Storyboard**-nak és továbblépünk. A folyamat után megjelenik az időszám, ennek segítségével fogunk vezérlőket és tulajdonságokat animálni.

Próbáljuk ki, hogy beállítunk egy időpillanatot (4-23 ábra), és utána megváltoztatunk egy tulajdonságot, mondjuk a négyzet színét. Ekkor létrejön az animáció a háttérben, amelyet itt helyben ki is próbálhatunk a lejátszás gombra kattintva.



4-22 ábra: Új storyboard létrehozása az animálás elkezdéséhez

Látva, hogyan működik Blendben az animáció, érdemes megnéznünk egy összetettebb példán keresztül azt, hogy milyen kód jön létre a háttérben. A következő mintában a 4-23 ábrán látott négyzet színét fogjuk megváltoztatni egy másodperc alatt, közben azt eltoljuk a **(100, 100)** koordinátájú pontba és megforgatjuk 360 fokban. A **ColorAnimation** segítségével egy vezérlő **Brush** típusú tulajdonságát tudjuk befolyásolni (kitöltés, szegély, betűszín, és így tovább). A **Storyboard** tulajdonságain keresztül kell beállítanunk, hogy melyik vezérlőt és melyik tulajdonságot szeretnénk animálni. Maga a **ColorAnimation** csak egy értéket vár, azt a színkódot, amelybe a jelenlegi színből kiindulva az animációval el szeretnénk jutni. A **Duration** tulajdonságnak megfelelő ideig színátmeneteket fogunk látni, amint az alapszínből áttérünk a megadott színre.



4-23 ábra: Egy animációs lépés elvégzése

A **DoubleAnimation** valamilyen lebegőpontos tulajdonságot tud animálni, az előzőekhez hasonló működéssel. Az alábbi példa ki van bővítve ún. **KeyFrame** alapú animációra, ami azt jelenti, hogy nem egy adott értéket szeretnénk az animáció segítségével a meghatározott idő alatt elérni, hanem a meghatározott időpillanatokban egy-egy konkrét értéket szeretnénk felvenni.

```
<UserControl.Resources>
  <Storyboard x:Name="Storyboard1">
    <ColorAnimation Duration="0:0:1" To="#FFCEFF2F"
      Storyboard.TargetProperty="(Shape.Fill).(SolidColorBrush.Color)"
      Storyboard.TargetName="rectangle" d:IsOptimized="True"/>

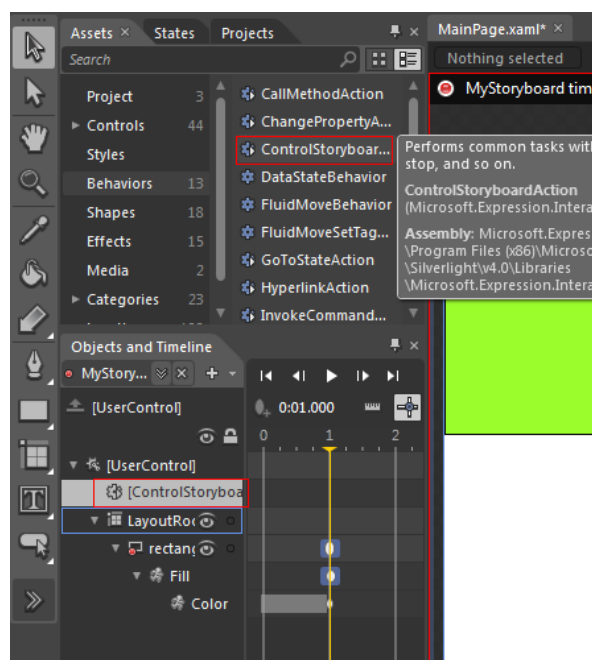
    <DoubleAnimationUsingKeyFrames
      Storyboard.TargetProperty=
        "(UIElement.RenderTransform).(CompositeTransform.TranslateX)"
      Storyboard.TargetName="rectangle">
      <EasingDoubleKeyFrame KeyTime="0:0:0.1" Value="0"/>
      <EasingDoubleKeyFrame KeyTime="0:0:2" Value="100"/>
    </DoubleAnimationUsingKeyFrames>

    <DoubleAnimationUsingKeyFrames
      Storyboard.TargetProperty=
        "(UIElement.RenderTransform).(CompositeTransform.TranslateY)"
      Storyboard.TargetName="rectangle">
      <EasingDoubleKeyFrame KeyTime="0:0:0.1" Value="0"/>
      <EasingDoubleKeyFrame KeyTime="0:0:2" Value="100"/>
    </DoubleAnimationUsingKeyFrames>

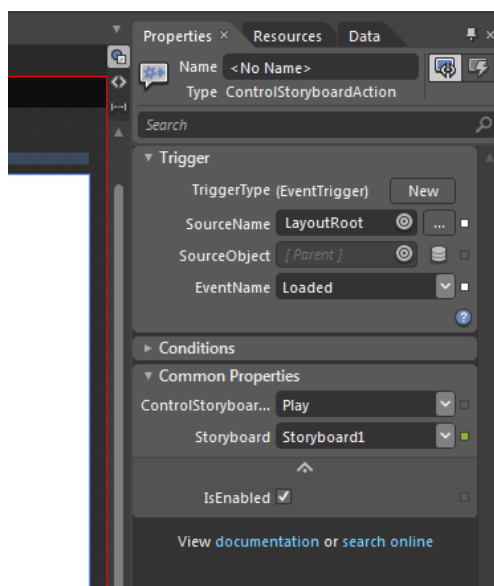
    <DoubleAnimationUsingKeyFrames
      Storyboard.TargetProperty=
        "(UIElement.RenderTransform).(CompositeTransform.Rotation)"
      Storyboard.TargetName="rectangle">
      <EasingDoubleKeyFrame KeyTime="0:0:0.1" Value="0"/>
      <EasingDoubleKeyFrame KeyTime="0:0:2" Value="360"/>
    </DoubleAnimationUsingKeyFrames>
  </Storyboard>
</UserControl.Resources>
<Grid x:Name="LayoutRoot" Background="White">
```

```
<Rectangle x:Name="rectangle" Fill="#FF2F2FFF"
    HorizontalAlignment="Left" Height="200" Stroke="Black"
    VerticalAlignment="Top" Width="200" RenderTransformOrigin="0.5,0.5">
    <Rectangle.RenderTransform>
        <CompositeTransform/>
    </Rectangle.RenderTransform>
</Rectangle>
</Grid>
```

Mielőtt a példát ki tudnánk próbálni, valamilyen eseményhez kell kötnünk az animáció elindítását. Erre két lehetőségünk van. Az egyik, hogy kódból meghívjuk a **Storyboard** objektum **Begin** metódusát. A másik lehetőség **Behavior**-ök használata a Blendben. Ekkor az Assets panelről az adott vezérlőre húzunk egy **ControlStoryboard** viselkedést, és beállítjuk, hogy melyik animációt és melyik esemény hatására szeretnénk lejátszani, ahogyan az a 4-24 és 4-25 ábrákon is látható.



4-24 ábra: Viselkedés alkalmazása egy vezérlőre



4-25 ábra: A ControlStoryboard viselkedés paraméterezése

Átmenetek és finomítások az animációban

Silverlight 3-ban jelent meg az ún. *easing function*, amely az animációk finomításában tud sokat segíteni. Ezekkel a matematikai formulákkal olyan viselkedéseket húzhatunk az animációkra, amelyekkel például annak elején és végén pattogást, rezgést vagy hullámozást — vagy bármilyen az easing function által definiált animációt — tudunk szimulálni, ezzel is sokkal természetesebb hatást kölcsönözve neki.

```
<StackPanel x:Name="LayoutRoot" Background="White">
  <StackPanel.Resources>
    <Storyboard x:Name="myStoryboard">
      <DoubleAnimation From="30" To="200" Duration="00:00:3"
        Storyboard.TargetName="myRectangle"
        Storyboard.TargetProperty="Height">
        <DoubleAnimation.EasingFunction>
          <BounceEase Bounces="2" EasingMode="EaseOut"
            Bounciness="2" />
        </DoubleAnimation.EasingFunction>
      </DoubleAnimation>
    </Storyboard>
  </StackPanel.Resources>

  <Rectangle x:Name="myRectangle" MouseLeftButtonDown="Mouse_Clicked"
    Fill="Blue" Width="200" Height="30" />
</StackPanel>
```

A fenti példában egy „pattogás” animációt adtunk meg, amely kettőt pattan és kizárólag akkor, amikor az animáció véget ért.

Összefoglalás

Ebben a fejezetben megnéztük azokat az alapvető megoldásokat, amelyeket a Silverlight nyújt számunkra felhasználói felületeink elkészítéséhez. Röviden megnéztük, hogyan tudunk multimédiás tartalmakat lejátszani, és hogyan tudjuk még interaktívabbá tenni alkalmazásunkat animációk segítségével.

5. Stílusok és testreszabhatóság a Silverlightban

A mai világ elvárásai rendkívül magasak az alkalmazásokat illetően, legyen szó webes vagy asztali alkalmazásokról. Nem kivétel ezek alól a felhasználói élmény sem. Egy szoftver felhasználói élményét (UX — user experience) sok tényező együttes működése és hatása határozza meg. Ezek közül az egyik legfontosabb és talán legszembetűnőbb a felhasználói felület (GUI). Ez az első frontvonal, amellyel alkalmazásunk majdani felhasználói először találkoznak. Ez lesz az ítélet első mércéje, ezen állhat vagy bukhat az alkalmazás sikere, elfogadottsága. Eleinte ez a tényező inkább csak a webes világot érintette, de a mai fejlett világban, ahol a technológia adott, az üzleti alkalmazások megjelenésével szemben támasztott követelmények is megváltoztak.

Ennek megfelelően a felhasználói felületet tervező szakemberek (designerek) igyekeznek egységes felületeket és stílusokat készíteni, más szóval *témákat*, amelyek az egész alkalmazás kinézetét és hangulatát alapvetően meghatározzák.

Korábbi Microsoft-os UI technológiák számtalan alkalommal okoztak a fejlesztők számára fejtörést rugalmatlan objektummodelljeik miatt, melyek miatt a testreszabhatóság szabadsága erős korlátok közé szorult.

Sztori: Mindannyian készítettünk már olyan alkalmazást, ahol több párhuzamos feladat aktuális állapotát kellett megjeleníteni. Ilyenek például a különböző letöltő kliensek is, amelyek többnyire egy listview jellegű vezérlőben mutatják, hogy az adott állomány letöltése éppen hol tart. Az esetek többségében nemcsak egy számmal jelzik a folyamat előrehaladását, hanem egy progressbarral is vizualizálják a százalékos értéket.

Windows Forms alatt kellett saját alkalmazásomba beépíteni egy ilyet. Sajnos közben kiderült, hogy a Windows Forms-ban elérhető **ListView** vezérlő csak bizonyos típusú vezérlőket hajlandó elfogadni egy oszlopban, és bizony a **ProgressBar** nem tartozik ezek közé. Csak komoly hackelések árán sikerült odakényszeríteni a vezérlőt. Ez a probléma Silverlight és WPF esetén egyszerűen fel sem merül azok rugalmas objektum modelljének köszönhetően!

Ebben a fejezetben megismerjük, hogy a Silverlight 4 miként támogatja az egységes felhasználói felületek kialakítását, valamint azt, hogy miként szabhatjuk teljesen át vezérlőinket különböző sablonok segítségével.

Erőforrások a Silverlightban

Az erőforrások szerepe

Tegyük fel, hogy választottunk egy **Brush** objektumot! Ezt az alkalmazásban több helyen szeretnénk szerepeltetni háttérkitöltésként. Persze mindenhova begépelhetjük annak definícióját beleégetve azt az alkalmazásba, de gondban leszünk, ha később cserélni szeretnénk azt. Az sem segít a helyzeten, ha a **Brush** netán komplexebb objektum, és nemcsak egy hexadecimális négyest (RGBA) kell a **Background** tulajdonságként meghatározni, hanem annál jóval többet. Például, ha színátmenetes hátteret szeretnénk, akkor használhatunk **LinearGradientBrush** vagy **RadialGradientBrush** objektumokat, attól függően, hogy milyen színátmenetet képzelünk el.

Ebből a szituációból egyértelműen adódik az a természetes igény, hogy ezt a **Brush** objektumot el tudjuk tárolni egy központi helyen és újra fel tudjuk használni az alkalmazás különböző pontjain. Ezzel rögtön definiáltuk is az erőforrások fogalmát.

Az *erőforrások* olyan tetszőleges típusú objektumok, melyeket nemcsak lokálisan, hanem több egymástól független helyen is fel kívánunk használni. Ilyen erőforrás objektum lehet például egy szín, egy adatobjektum, egy vezérlő sablon, egy string, és így tovább.

A *StaticResource* objektum

A Silverlightban ennek a fogalomnak megfelelője a **StaticResource**, azaz a statikus erőforrás. Na de miért statikus? Mint azt már számtalanszor hallottuk, a Silverlight a WPF (*Windows Presentation Foundation*) leszármazottja (WPF Everywhere volt a korai Silverlight projekt kódneve). A WPF-ben két típusú erőforrás létezik, a statikus és a dinamikus. Ez utóbbi a Silverlightban jelen pillanatban nincsen. Ezért kapta a **Static** prefixet az erőforrás-hivatkozás. Na és mit jelent, hogy statikus? A statikus erőforrás egyrészről azt jelenti, hogy már fordításidőben bekötésre kerül az erőforrás által meghivatkozott objektum, másrészről azt, hogy az erőforrás nem cserélhető futásidőben. Azaz, ha egy **Background** tulajdonsághoz hozzákötünk egy **SolidColorBrush** erőforrás objektumot, akkor futás közben ez az erőforrás nem cserélhető le. (Természetesen a **Background** tulajdonsághoz hozzárendelhetünk egy másik erőforrást.) Persze ez nem jelenti azt, hogy nem lehetne futásidőben megváltoztatni az erőforrás segítségével a háttérszínt, ugyanis a **SolidColorBrush** objektum tulajdonságai módosíthatók, így például a **Color** új értéket kaphat, ami új háttérkitöltéshez vezet.

Az erőforrások hozzáférhetősége

Felmerül az a kérdés, hogy hol definiálhatunk erőforrásokat. A válasz egyszerű: a **Resources** szekcióban. A **Resources** tulajdonság a **FrameworkElement** ősszármaztatott osztályban került definiálásra, értéke egy **IDictionary<string, object>** típusú objektum. Ennek megfelelően minden vezérlő, panel, sőt még az **Application** objektum is rendelkezik egy ilyen gyűjteménnyel. Így az erőforrásokat több szinten lehet definiálni. Három fő szintet emelnék ki:

- Application szint
- UserControl szint (teljes oldal)
- Vezérlő szint (lokálisan)

Az, hogy az adott erőforrást melyik szinten definiáljuk, egyértelműen meghatározza, hogy mely objektumok számára lesz elérhető. Ennek megfelelően, ha az **Application** objektum **Resources** gyűjteményében helyezünk el egy erőforrást, akkor az az alkalmazásban mindenhol hivatkozható lesz. Ha kizárólag az adott oldalon szeretnénk elérhetővé tenni a kérdéses erőforrást, és nem szeretnénk, hogy más oldalon is elérhető legyen, akkor az adott oldalt reprezentáló **UserControl** objektum **Resources** szekciójában kell elhelyezni azt. Ha egy erőforrást csak egy panelen vagy vezérlőn belül definiálunk, akkor a vezérlőfában a panellel vagy a vezérlővel egy szinten levő egyéb vezérlők számára erőforrásunk láthatatlan lesz. Azaz, egyfajta öröklődési láncról beszélhetünk.

Az erőforrások használata

Erőforrást a következő kódrészlet segítségével definiálhatunk:

```
<UserControl.Resources>
  <SolidColorBrush x:Key="bgColor" Color="Blue"/>
  ...
</UserControl.Resources>
```

Ezen a **UserControl**-on belül ez az erőforrás minden objektum számára látható és elérhető. Hivatkozni az **x:Key** attribútum értékén keresztül lehet rá a **StaticResource** nyelvi kiterjesztés (*markup extension*) használatával.

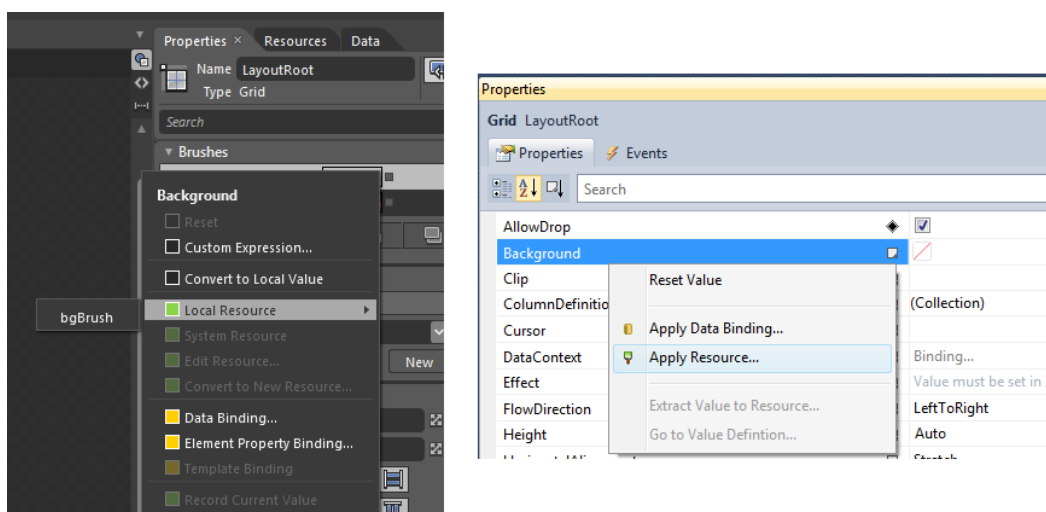

```
// Szintaktika
<Objektum PropertyNev="{StaticResource resourceKey}"/>

// Hivatkozás xaml-ből
<Grid Background="{StaticResource bgColor}">
    ...
</Grid/>

// Hivatkozás C#-ból
SolidColorBrush brush = this.Resources["bgColor"] as SolidColorBrush;
```

A **this** kulcsszónak itt a **UserControl**-ra kell vonatkoznia (pl. a mögötte lévő kód egy eseménykezelőjében vagyunk, az adott **UserControl** osztályban). A **Resources** gyűjtemény értéke mindig **object** típusú, ezért van szükség a típuskonverzióra (**as** operátor). A **Key** attribútum értékének természetesen egyedinek kell lennie.

Erőforrások bekötésére a tervezőeszközök segítségével is van lehetőség. Az Expression Blend esetén erőforrást az adott tulajdonság melletti Advanced Property Options segítségével köthetünk be. A felugró menüből a megfelelő Local Resources szekcióból lehet kiválasztani a kívánt erőforrást. A Visual Studio 2010 esetén a Properties ablakban található az Advance Properties menü, ahol az Apply Resource opciót választva köthetjük be a megfelelő erőforrást (5-1 ábra).



5-1 ábra: Erőforrás bekötése Blendből, illetve Visual Studio 2010-ből

Az erőforrás fájlok (ResourceDictionary)

Ahogy folyamatosan haladunk előre a Silverlight megismerésében, és írjuk saját alkalmazásainkat, látni fogjuk, hogy az XAML több, mint 90%-a erőforrás definíciókból fog felépülni. A karbantartásuk így egyre nagyobb gondot fog okozni. Annak ellenére, hogy az elérhetőségi szintekkel és különböző resource szekciókkal sokáig trükközhetünk, nagyon gyorsan világossá fog válni, hogy további strukturálásra van szükség.

Az alkalmazáshoz épített stílusok újrafelhasználhatósága előbb-utóbb fontos cél lesz. A saját témák építése és azok újbóli felhasználhatósága is fontos követelménnyé válik.

Ezekre a problémákra szolgál gyógyírként a **ResourceDictionary**. Ezt az objektumot egy külső XAML fájlban definiáljuk, és a felhasználás során meghivatkozzuk. Ilyen XAML állomány lehet például a **ResourceDictionary1.xaml**, az alábbi tartalommal:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <SolidColorBrush x:Key="bgColor" Color="Blue"/>
  ...
</ResourceDictionary>
```

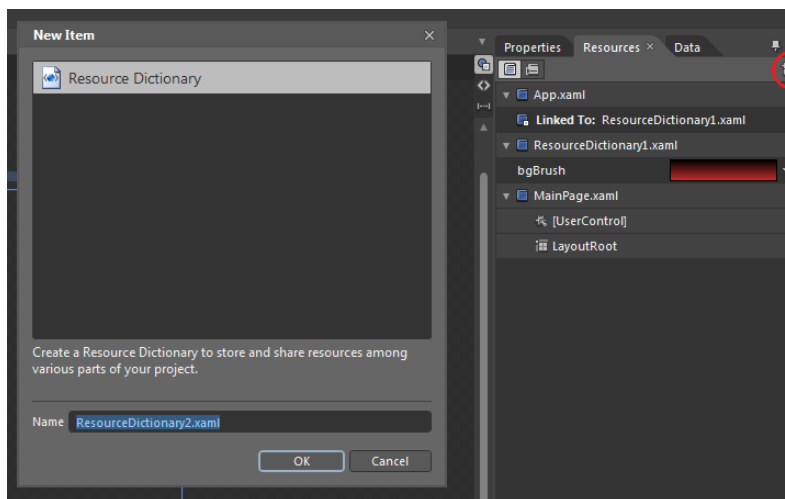
Ezt az erőforrás-könyvtárat használó **App.xaml** tartalma az alábbi módon nézhet ki:

```
<Application
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="ResourcesDemo.App">
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="ResourceDictionary1.xaml"/>
        <ResourceDictionary Source="ResourceDictionary2.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

Application szintre emeltük be a **ResourceDictionary1.xaml** és a **ResourceDictionary2.xaml** tartalmát. Így a fenti **bgColor** erőforrás az alkalmazás tetszőleges pontján meghivatkozható.

Ezek a XAML fájlok átvihetők más projektekbe, így a témáinkat újra és újra felhasználhatjuk.

A **ResourceDictionary** objektumok létrehozását és kezelését az Expression Blend is támogatja. A Resources ablak a jobb felső sarokban található azon funkció, amellyel új erőforrás könyvtárat lehet létrehozni, valamint ebben az ablakban tekinthetjük meg, hogy milyen erőforrásokat definiáltunk és hol. Az erőforrásokat itt „drag-and-drop” segítségével szabadon mozgathatjuk az erőforrás szintek és a **ResourceDictionary**-k között. Az 5-2 ábrán jól látható, hogy már van egy **ResourceDictionary1.xaml** betöltve az **Application Resources** szekciójában, és egy **bgBrush** erőforrás szerepel benne. Ezt az erőforrást most könnyedén átmozgathatnánk például a **UserControl** vagy a **LayoutRoot Resources** szekciójába. Az ábrán látható továbbá, hogy egy új **ResourceDictionary** kerül felvételre **ResourceDictionary2.xaml** néven.



5-2 ábra: *ResourceDictionary* objektumok és erőforrások csoportosítása az Expression Blendben

Stílusok

A stílusok szerepe

A témák kialakításának egy nagyon fontos építőköve a tulajdonságértékeken való osztózás. Például panelek esetén közös háttérszín, **Border** objektumok esetén közös lekerekítési mérték, feliratok esetén azonos betűtípus és méret. Képzeljük el a következőt! Az alkalmazásunkban mindenhol az adott oldal címe 32 pt méretű, SegoeUI típusú, félkövér betűkészlet. A felugró, beúszó panelek keretének háttere egységesen szürke-fehér lineáris átmenetű, azok 15-ös lekerekítéssel rendelkeznek, a keret vastagsága 1 pixel, színe sötétszürke. Az erőforrások ismeretében természetesen adódik, hogy ezeket a jellemzőket erőforrásként kell definiálni, hogy azok újrafelhasználhatók legyenek. Ezúttal azonban van még egy extra igényünk is! A csoportosíthatóság. Más tulajdonságok érvényesek az oldalak főcímére és mások a felugró panelekre. A csoporton belüli összes tulajdonságot egyszerre akarjuk alkalmazni az adott objektumokon. Ezen a ponton kerülnek képbe a *stílus* objektumok.

A stílusokra legegyszerűbben úgy érdemes gondolni, hogy azok valójában nem mások, mint „tulajdonság halmazok”. Fogunk egy rakás közös jellemzőt, és egyetlen stílusba gyömöszöljük őket, majd magát a stílus objektumot helyezzük el erőforrásként.

Stílusokat a Silverlightban a **Style** objektum segítségével lehet definiálni, miközben a **TargetType** tulajdonságot beállítjuk arra a típusra, amelyre a stílus készül. Stílusok esetében az egyes tulajdonságok beállítása **Setter**-ek segítségével történik, ahogyan az alábbi példában is látható:

```
// Szintaktika
<Style x:Key="[resourceKey]" TargetType="[KontrolTipus]">
    <Setter Property="[PropertyNev]" Value="[Ertek]"/>
    ...
</Style>

//Példa
<UserControl.Resources>
    <Style x:Key="myBorderStyle" TargetType="Border">
        <Setter Property="CornerRadius" Value="15"/>
        <Setter Property="Background">
            <Setter.Value>
                <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
                    <GradientStop Color="White" Offset="1"/>
                    <GradientStop Color="#FF616161" Offset="0"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
        <Setter Property="BorderThickness" Value="1"/>
        <Setter Property="BorderBrush" Value="#FF383838"/>
    </Style>
</UserControl.Resources>
```

Ha a tulajdonságokhoz tartozó érték komplex és így nem lehet azt egyetlen stringben leírni (lásd pl. a **Background** tulajdonságot), akkor a **Setter** elem **Value** értéke „kinyitható”, azaz attribútum helyett XML elemként használható.

A **Style** tulajdonságot a **FrameworkElement** őssztály definiálja, így például minden vezérlő és panel rendelkezik vele. Nem maradt más dolgunk, mint alkalmazni a stílust:

```
// Stílus alkalmazása xaml-ból
<Grid x:Name="LayoutRoot">
    <Border x:Name="myBorder" .... Style="{StaticResource myBorderStyle}" />
</Grid>
```

```
// Stílus alkalmazása dinamikusan C#-ból  
Style style = this.Resources["myBorderStyle"] as Style;  
myBorder.Style = style;
```

Stílusok öröklődése

Nem kell mindig minden stílust a nulláról felépíteni. Számos esetben építkezhetünk valamilyen alapstílusból kiindulva, amelyből később leszármaztatva specifikusabb stílusokat lehet készíteni. A Silverlight 3 óta a **Style** objektum rendelkezik a **BasedOn** attribútummal, amely lehetővé teszi a hivatkozást egy ősstílusra, így szolgáltatva közös alapot más stílusok számára.

```
<Style x:Key="errorPopupBorderStyle" TargetType="Border"  
      BasedOn="{StaticResource myBorderStyle}">  
    <Setter Property="BorderBrush" Value="Red"/>  
    ...  
</Style>
```

Ez a stílus rendelkezik minden olyan tulajdonsággal, amivel a **myBorderStyle**, de felüldefiniálja a **BorderBrush** értéket, ami itt most a piros szín lesz. Ezenkívül rendelkezhet még egyéb tulajdonság-beállításokkal is.

Ezzel most már lassan szemünk elé tárul a tulajdonságok felüldefiniálásának sorrendje, prioritása is. Az ősstílus tulajdonságait felülírják a leszármazott stílusok tulajdonságai. Bármely stílus beállítását felüldefiniálja a közvetlenül a vezérlőn definiált, lokálisan meghatározott érték.

Implicit stílusok

Felmerül azonban egy kellemetlen kérdés. Ha én azt szeretném, hogy egy elem rendelkezzen egy stílussal, akkor mindig manuálisan meg kell hivatkoznom a stílust és bekötnöm a megfelelő elemhez? Szerencsére Silverlight 4 óta nem. Ebben a verzióban ugyanis megjelentek az implicit stílusok. Jelen pillanatban, amikor egy **Style** elemet definiálunk, meghatározzuk a **Key** és a **TargetType** attribútumokat. A **Key** meghatározása a **Resources** gyűjtemény miatt szükséges — hiszen az kulcs-érték párokat tárol. A Silverlight 4-től kezdve azonban, ha nem adjuk meg a stílus **Key** tulajdonságát, hanem csak a **TargetType** jellemzőt határozzuk meg, akkor az összes alkalmazható elemre a stílus beállításra kerül.

Mit jelent az, hogy alkalmazható elem? Két szempontot kell figyelembe vennünk. Egyrésztől kérdés a hozzáférhetőség, hiszen számít, hogy a stílust milyen szinten definiáltuk, hiszen csak a stílushoz hozzáférő elemek alkalmazhatják azt implicit módon. Másrésztől a típus is számít, hiszen ha a **TargetType** értéke **Button**, akkor csak a gombokra lesz érvényes a stílus, de ha a **TargetType** értéke mondjuk **Control**, akkor az összes vezérlőre érvényes lehet implicit módon.

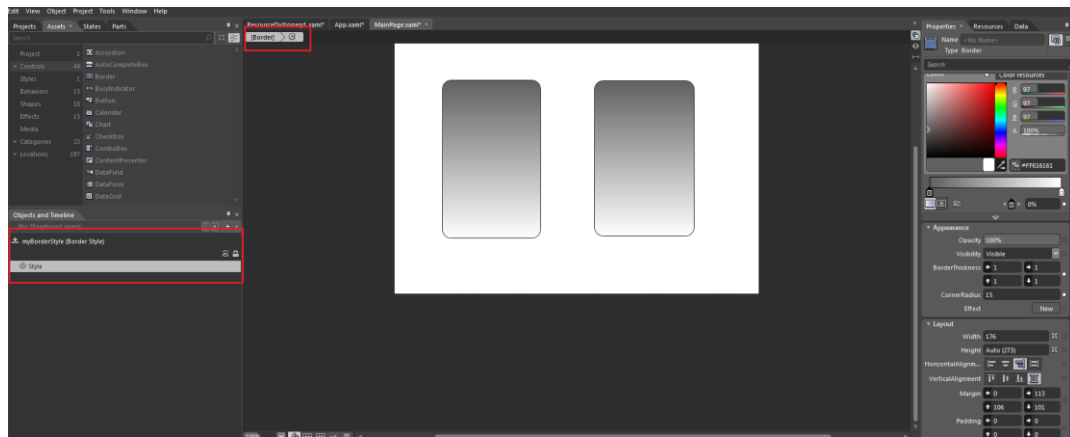
Most már talán jól érthető, hogy a közhiedelemmel ellentétben a stílusok a Silverlightban nem a testreszabhatóságot, hanem az újrafelhasználhatóságot támogatják! Nem vezetnek be semmilyen új koncepciót vagy modellt, kizárólag a meglevő tulajdonságok egyfajta egységbezárását és disztribúcióját támogatják. Szerepüket azonban lebecsülnünk sem szabad, hiszen segítségükkel a saját témák kialakítása az alkalmazáson belül — akár hordozható formában is — szinte gyerekjátékká válik.

Stílusok definiálására, szerkesztésére az Expression Blendben is van lehetőség. Az adott vezérlőt kiválasztva, majd az Object menüre kattintva a Styles menüpontban több lehetőségünk is van:

- Az Edit Current azt jelenti, hogy már van egy stílus definiálva az adott objektumon, és azt szeretnénk tovább szerkeszteni.
- Az Edit a Copy azt jelenti, hogy a vezérlő rendelkezik már egy stílussal, de ebből a stílusból inkább egy másolatot készítenénk, és azt szerkesztenénk tovább, így érintetlenül hagyva azokat a vezérlőket, amelyek az eredeti stílust használják.

- Create Empty-vel nulláról építhetünk fel egy teljesen új stílust.
- Apply Resource segítségével pedig egy Resource-ként definiált stílust alkalmazhatunk az adott elemen.

Stílusszerkesztés módba lépve az Objects and Timeline ablak „Style módba” vált, azaz a vizuális fa többé már nem látható. Ezen nincs miért csodálkoznunk, egy stílus belsejében vagyunk éppen, itt nincs vizuális fa. Figyelem: bármilyen tulajdonságot is állítunk most be a Properties ablak vagy akár a kurzor segítségével a tervezőfelületen, minden módosítás a stílus részévé fog válni, azaz egy **Setter** kerül be az adott beállításra vonatkozóan (5-3 ábra)! Az ábrán látható két **Border** ugyanazon a stílus objektumon osztozik. Az Objects and Timeline ablakban egyetlen Style objektum szerepel, ezzel is jelezve, hogy szerkesztés módban vagyunk, továbbá a designer felület fölött egy „breadcrumbs” menü is mutatja, hogy a Border objektum stílus elemében tartózkodunk. Bármit is állítunk be most a jobb oldali properties menüben, az a myBorderStyle stílus részét fogja képezni.



5-3 ábra: Stílusok definiálása az Expression Blendben

A vezérlők testreszabása

A ControlTemplate-ek szerepe

Stílusok segítségével közös jellemzőket alkalmazhatunk a különböző vezérlőkön, de ezzel valódi testreszabást nem tudunk megvalósítani. A tulajdonságok egyike azonban kiemelkedik az összes közül. Ez a **Control** ósoszvány **Template** tulajdonsága, amely egy **ControlTemplate** típusú objektum. A Silverlight-ban minden vezérlő ún. primitívekből épül fel. A legösszetettebb vezérlő is visszavezethető **Rectangle**, **Ellipse**, **Path**, **Border** és **Panel** objektumokra. Gyakorlatilag minden vezérlő kinézete ezekből az alakzatokból áll össze. Sokat lehet hallani, hogy a Silverlightban a vezérlők kinézet nélküliek (*lookless controls*), szinte csak funkcionalitásból állnak. Így egy gombra nem úgy gondolunk, hogy egy téglalap, ami benyomódik, ha rákattintanak, hanem egy „objektum, amire kattintani lehet”. A Silverlightban a vezérlők logikája és a hozzájuk tartozó megjelenés szinte teljesen elválnak egymástól, ez egy lazán csatolt kapcsolat. (A csatolás pontos módjáról a Custom Controlok fejezetben beszélünk.)

Ennek a koncepciónak köszönhetően az a primitív elemekből álló vizuális fa, ami a vezérlő megjelenítését szolgálja, szabadon testreszabható, vagy akár teljes egészében ki is cserélhető, ezáltal teljesen új külsőt kölcsönözve a vezérlőinknek.

ControlTemplate-ek definiálása.

A **Template** is csak egy, a többihez hasonló tulajdonság, így nyugodtan szerepelhet egy stílus részeként, amelyben az kerül kifejtésre egy **Setter**-ben. Akár külön erőforrásként is definiálhatjuk, sőt akár magán a szerkesztés alatt álló objektumon is készülhet közvetlenül.

Nézzünk egy egyszerű példát, készítsünk egy saját gombot!

```
<Button Content="Hello" Width="120" Height="40" Background="Red">
  <Button.Template>
    <ControlTemplate TargetType="Button">
      <Rectangle Fill="Blue"
        RadiusX="15" RadiusY="15"
      />
    </ControlTemplate>
  </Button.Template>
</Button>
```

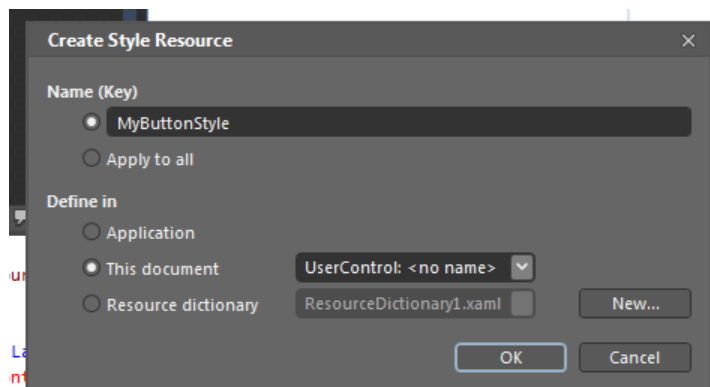
Ezzel készítettünk egy sarkainál lekerekített kék gombot, amit az 5-4 ábra mutat be.



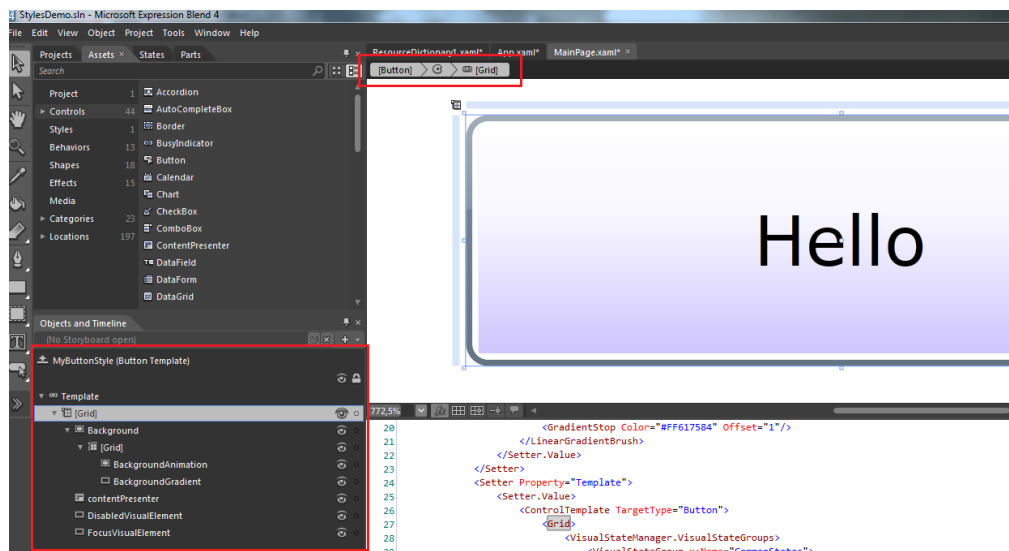
5-4 ábra: Első ControlTemplate-ünk

Egy **ControlTemplate** létrehozása az Expression Blend segítségével is lehetséges. Az adott vezérlőn egy jobb klikket követően válasszuk az Edit Template menüpontot! Itt több módon is továbbhaladhatunk:

- Az Edit Current azt jelenti, hogy már definiáltunk egy sablont az adott objektumon, és azt szeretnénk tovább szerkeszteni.
- Az Edit a Copy azt jelenti, hogy a vezérlő rendelkezik már egy sablonnal, de arról inkább egy másolatot készítenénk, és azt szerkesztenénk tovább, így érintetlenül hagyva azokat a vezérlőket, amelyek az eredeti stílust használják. A sablont alapvetően egy **Style** objektum belsejében fogja a Blend eltárolni, és ennek létrehozásakor fel is dobja majd a Create Style Resource dialógust, ahol kiválaszthatjuk az erőforrás nevét, továbbá az elérhetőségi szintjét (5-5 ábra). Ennek az opciónak a különlegessége, hogy első alkalommal az ún. „default chrome”, azaz a vezérlő eredeti vezérlőfájának felépítését is megmutatja (5-6 ábra).
- A Create Emptyvel egy üres sablonból kiindulva építhetünk egy teljesen újat.
- Az Apply Resource segítségével egy **Resource**-ként definiált sablont alkalmazhatunk az adott elemen.



5-5 ábra: Új stílus létrehozása és erőforrásként történő mentése



5-6 ábra: „Edit a Copy” választása új ControlTemplate létrehozásakor

A gomb a funkcionalitását illetően egyébként teljes értékű, a **Click** esemény lefut rajta, amint rákattintunk a téglalapra — ebben nincsen hiba. A vizuális visszajelzést illetően viszont bőven találhatunk kivetnivalót. Nem reagál arra, hogyha az egér a gomb fölött van (**MouseEnter**), és ami azt illeti, azt sem nagyon látni, ha rákattintunk a gombra (**Click**). További hiba, hogy a „Hello” szöveg annak ellenére, hogy tartalomként beállításra került, nem jelenik meg, és a háttér sem piros, hanem kék. Nézzük meg közelebbről ezeket a problémákat sorjában!

Ismerkedés a ContentPresenter-rel

Az első és legégetőbb probléma, hogy a **Button** vezérlő **Content** tulajdonságához rendelt érték nem jelenik meg. Ahhoz, hogy ennek okát megértsük, képzeljük magunkat a Silverlight helyébe! A **ControlTemplate** belsejében definiálunk valamilyen komplex struktúrát, például sok **Panel**-t, **Rectangle**-t, **Border**-eket, majd ezek után — jelen esetben nekünk — kellene a megfelelő helyre berakni a „Hello” szöveget, azaz a nyomógombunk **Content** tulajdonságát. Hogyan döntjük el, hogy a komplex vezérlőfában hol helyezzük el a szöveget? Segítség nélkül sehogyan.

Szerencsénkre a segítségünkre siet a **ContentPresenter** objektum! A koncepció a következő:

A komplex vezérlőfában jelöljük meg, hogy hova kerüljön a **Content** tulajdonságba helyezett tartalom. Azaz lényegében egy jelölést — egy ún *placeholder*t helyezünk el. Esetünkben, a **ContentControl** objektumoknál ez egy **ContentPresenter**, **ItemsControl** objektumoknál pedig az **ItemsPresenter**. Ha a Silverlight talál egy **ContentControl** belsejében egy **ContentPresenter**-t, akkor oda rakja be a tartalmat. Ha nem talál ilyet, akkor nem rakja sehova a **Content** tulajdonság tartalmát, egyszerűen kimarad ez a lépés. **ItemsControl** objektumok esetén az **Items** tulajdonság tartalma jelenik meg az **ItemsPresenter** helyén.

Nincs más dolgunk, mint egy kicsit áttervezni a **ControlTemplate** elemet, és elhelyezni benne egy **ContentPresenter**-t:

```
<ControlTemplate TargetType="Button">
  <Grid>
    <Rectangle Fill="Blue" RadiusX="15" RadiusY="15"/>
    <ContentPresenter HorizontalAlignment="Center"
      VerticalAlignment="Center"/>
  </Grid>
</ControlTemplate>
```


Ezzel a módosítással már megjelenik a gombunkban a „Hello” felirat. De bármi mást is helyezünk el a **Content** tulajdonságban, az a gomb közepén meg fog jelenni. A tartalom középre igazítását a **ContentPresenter** középre igazításával értük el.

A *TemplateBinding* fogalma

Második problémánk az volt, hogy bár a **Button** vezérlő háttérszínét pirosként határoztuk meg, a gombunk továbbra is kékszíni maradt. Ha egy picit jobban belegondolunk, akkor már sejtjük, hogy miért van ez így. A **Rectangle** alakzat **Fill** tulajdonságát kékként határoztuk meg. Ismét csak arról van szó, hogy a Silverlight nem tudja azt, hogy a **ControlTemplate**-ben felépített komplex vezérlőfa melyik elemének melyik tulajdonsága fogja meghatározni a teljes **Button** hátterét. Mivel magától ezt nem tudhatja, most is sügnünk kell neki! A sugásban a **TemplateBinding** markup extension siet segítségünkre. A dolgunk egyszerű. Meg kell mondanunk, hogy a **Rectangle** alakzat **Fill** tulajdonsága legyen hozzákötve a **Button** vezérlő **Background** tulajdonságához. Ez azt jelentené, hogy ha valaki azt mondja, hogy a gomb háttere legyen vörös, akkor a **Rectangle Fill** tulajdonsága magától a vörös színt veszi fel. Végezzük el a módosítást:

```
// Szintaktika
// <BelsőObjektum Property="{TemplateBinding KülsőObjektumProperty}"/>

<ControlTemplate TargetType="Button">
    <Grid>
        <Rectangle Fill="{TemplateBinding Background}" RadiusX="15" RadiusY="15"/>
        <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>
    </Grid>
</ControlTemplate>
```

A **Binding** mechanizmus jelentősen leegyszerűsíti az ilyen feladatokat. Ezen minta alapján tovább javíthatjuk a kódunkat, hogy a **Content** pozícióját a **Button**-on definiált tulajdonságok segítségével tudjuk meghatározni.

```
<ControlTemplate TargetType="Button">
    <Grid>
        <Rectangle Fill="{TemplateBinding Background}" RadiusX="15" RadiusY="15"/>
        <ContentPresenter
            HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
            VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
    </Grid>
</ControlTemplate>
```

Ezzel a módosítással a **Content** tulajdonság által reprezentált tartalmat a **Button**-on belül a vezérlő **HorizontalContentAlignment** és a **VerticalContentAlignment** tulajdonságainak segítségével lehet pozicionálni.

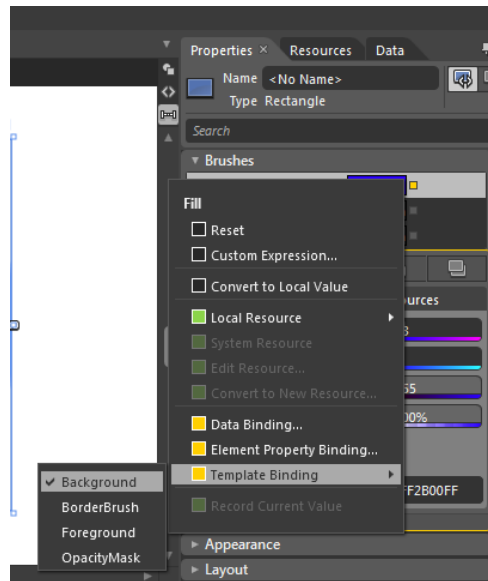
TemplateBinding-ot az Expression Blendben is beállíthatunk az Advanced Property Options segítségével, ahogy azt az 5-7 ábra mutatja. Az ábrán az látható, hogy a **Rectangle** alakzat **Fill** tulajdonságát kötjük a **Button** elem **Background** tulajdonságához.

Vizuális állapotok

A harmadik problémánk az volt, hogy a gombunk nem ad semmiféle vizuális visszajelzést a hozzá kapcsolódó történésekről. Így például nem jelzi, hogy éppen fölötte van az egér, fókuszban van, rákattintottak, és így tovább. Jól tudjuk, hogy a beépített — alapértelmezett — **ControlTemplate**-ek képesek erre is. Vajon mi hiányzik a miáltalunk definiált sablonokból? Hogyan oldhatnánk meg ezt a problémát?

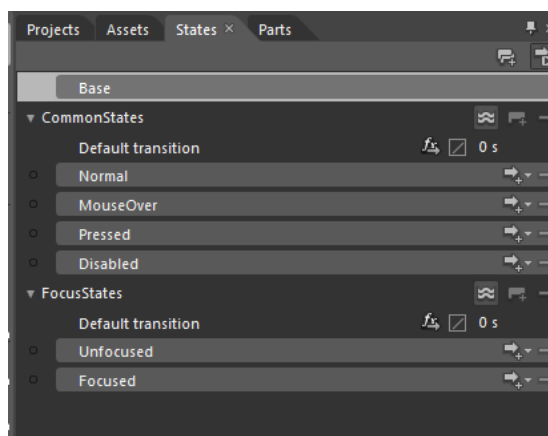
Első ötletünk az lehetne, hogy eseménykezelőket definiálunk a **MouseEnter** és a **MouseLeave** eseményekre. Ez azonban több szempontból is rossz ötlet lenne! Az első gond, hogy ebben az esetben a

vizualitáshoz kötnénk hozzá a logikát, vagyis nem lenne független a vezérlő megjelenése és a kapcsolódó prezentációs logika. A második problémánk, hogy **ControlTemplate**-ekben nem is definiálhatunk eseményeket a XAML jelölés segítségével. Így legalább nem fenyeget bennünket a megjelenítés és a hozzá kapcsolódó logika összekötésének veszélye.



5-7 ábra: TemplateBinding alkalmazása a Rectangle Fill tulajdonságán

A megoldás a Silverlightban a vizuális állapotok alkalmazása. A vezérlők készítője — jelen esetben a Microsoft — a vezérlők készítésekor számos vizuális állapotot definiált. A vizuális állapotok közötti váltást a megfelelő események bekövetkeztekor a vezérlők maguk végzik el. Nekünk csupán annyi a dolgunk, hogy meghatározzuk, hogy egy adott vizuális állapot elérésekor milyen animáció zajlik le. Vagyis, animációk segítségével definiáljuk, hogy egy adott állapotban hogyan néz ki a vezérlőnk. Az állapotváltások közötti vizuális átmenetet pedig az animációk interpolációs mechanizmusai határozzák meg. Az 5-8-as ábrán látható, miként biztosítja az Expression Blend a vizuális állapotok közötti váltást tervezés nézetben.



5-8 ábra: Egy Button objektumhoz tartozó vizuális állapotok megjelenítése az Expression Blendben

A vizuális állapotokat (**VisualState**) csoportokba soroljuk (**VisualStateGroup**) annak érdekében, hogy ne legyenek állapotütközések. Ez azt jelenti, hogy egy **VisualStateGroup**-on belül egyszerre csak egy **VisualState**-ben lehet a vezérlő. Ennek megfelelően például a **CommonStates VisualStateGroup**-on belül vagy **Pressed**, vagy **MouseOver** állapotban vagyunk, hiába tűnik úgy, hogy lehetne a két állapot

között átfedés, ezek valójában egymást kizáró állapotok. Azonban a gomb lehet egyszerre **Focused** és **MouseOver** állapotban, mivel a két állapot külön **VisualStateManager**-ban szerepel.

A Silverlight ellenőrzi, hogy az adott vizuális állapot a **ControlTemplate**-ben definiált-e. Amennyiben igen, akkor a megfelelő esemény bekövetkeztekor „bebillenti” a vizuális állapotot és lejátssza a kapcsolódó animációt. Amennyiben nincs ilyen vizuális állapot definiálva, nem történik semmi.

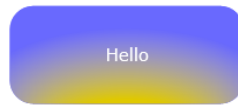
A fentieknek megfelelően két dolgot kell végrehajtanunk. Definiálnunk kell a vizuális állapotokat, illetve animálnunk a **ControlTemplate**-et.

Egészítsük ki a sablonunkat egy objektummal, amit érdemes lesz animálni!

```
<ControlTemplate TargetType="Button">
  <Grid>
    <Rectangle Fill="Blue" RadiusX="15" RadiusY="15" />
    <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>
    <Rectangle x:Name="mouseOverRect" RadiusX="15" RadiusY="15" Opacity="0">
      <Rectangle.Fill>
        <RadialGradientBrush GradientOrigin="0.5,1.633"
          RadiusY="0.912" RadiusX="0.912">
          <GradientStop Color="#FFE1CA00" Offset="0.331"/>
          <GradientStop Color="#69FFFFFF" Offset="0.689"/>
        </RadialGradientBrush>
      </Rectangle.Fill>
    </Rectangle>
  </Grid>
</ControlTemplate>
```

Ebben a kódrészletben a **mouseOverRect** elnevezésű **Rectangle** objektumot fogjuk animálni. Az **Opacity** tulajdonság alapértelmezetten, azaz „Normal” állapotban 0, tehát nem látszik. Ha bekövetkezik a **MouseEnter** esemény, a vezérlőnk belép a **MouseOver** vizuális állapotba (ezt még definiálnunk kell), és egy **Storyboard** segítségével megjeleníti a **mouseOverRect** elnevezésű **Rectangle** objektumot (5-9 ábra).

```
Vizuális állapotok definiálása XAML-ben:<ControlTemplate TargetType="Button">
  <Grid>
    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup x:Name="CommonStates">
        <VisualState x:Name="Normal"/>
        <VisualState x:Name="MouseOver">
          <Storyboard>
            <DoubleAnimation Duration="0:0:0.3" To="1"
              Storyboard.TargetProperty="(UIElement.Opacity)"
              Storyboard.TargetName="mouseOverRect"
              d:IsOptimized="True"/>
          </Storyboard>
        </VisualState>
        <VisualState x:Name="Pressed"/>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <Rectangle Fill="Blue" RadiusX="15" RadiusY="15"/>
    <Rectangle x:Name="mouseOverRect" RadiusX="15" RadiusY="15" Opacity="0">
      <Rectangle.Fill>
        <RadialGradientBrush GradientOrigin="0.5,1.633"
          RadiusY="0.912" RadiusX="0.912">
          <GradientStop Color="#FFE1CA00" Offset="0.331"/>
          <GradientStop Color="#69FFFFFF" Offset="0.689"/>
        </RadialGradientBrush>
      </Rectangle.Fill>
    </Rectangle>
    <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>
  </Grid>
</ControlTemplate>
```



5-9 ábra: Gombunk a *MouseOver* vizuális állapotban

Habár a vezérlő támogat még egy **Disabled** állapotot is, tekintve, hogy a példában nem kívánom felhasználni, így nemes egyszerűséggel elhagytam azt. Egy gondosan megírt vezérlőnek ez nem okozhat problémát.

A példát futtatva érdekes megállapításra juthatunk. A **MouseEnter** esemény bekövetkezésekor valóban megjelenik a **mouseoverRect** objektum egy rövid, kellemes animációval, amely mindössze 3 tizedmásodpercig tart. A **MouseLeave** esemény bekövetkezésekor visszabillen a vezérlőnk a **Normal** állapotba, anélkül, hogy ahhoz mi explicit módon animációt definiáltunk volna. Ez azt jelenti, hogy a Silverlight megjegyzi a tulajdonságok előző értékét, majd amikor vissza kell térni az aktuálisat megelőző állapotba, könnyedén elő tudja állítani azt. A **Normal** állapotba való átmenet most egyik pillanatról a másikra történik meg, mindenféle interpoláció nélkül, gyakorlatilag nulla időtartam alatt. Ezt a jelenséget két különböző módon is ki tudjuk küszöbölni.

Az egyik lehetőségünk, hogy készítünk egy ugyanolyan **Storyboard** objektumot, mint ami a **MouseOver** állapot belsejében van, csupán a **To** értéket határozzuk meg 0-ban, majd elhelyezzük a **Storyboard**-ot a **Normal** állapot belsejében. Ennek a megközelítésnek semmilyen hátránya nincs, talán csak annyi, hogy extra munkát igényel. Ugyanakkor ezt a módszert sokkal inkább alapértelmezett vizuális átmenetek felüldefiníálására szokták használni. Melyek is ezek az alapértelmezett vizuális állapotok? A második módszer éppen őket használja ki.

Vizuális állapotátmenetek

A másik lehetőségünk tehát az, hogy meghatározunk egy alapértelmezett átmenetet a teljes **VisualStateManager**-ra. Ezen az átmeneten beállíthatunk ún. **EasingFunction** viselkedést, illetve **Duration** értékeket, amelyek ettől kezdve a teljes **VisualStateManager**-ra értelmezettek lesznek. Természetesen, az állapotok belsejében definiált specifikus átmenetek, **Storyboard**-ok felülírják ezeket az alapértelmezéseket.

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="CommonStates">
    <VisualStateGroup.Transitions>
      <VisualTransition GeneratedDuration="0:0:0.3">
        <VisualTransition.GeneratedEasingFunction>
          <CubicEase EasingMode="EaseIn"/>
        </VisualTransition.GeneratedEasingFunction>
      </VisualTransition>
    </VisualStateGroup.Transitions>
    <VisualState x:Name="Normal"/>
    <VisualState x:Name="MouseOver">
      <Storyboard>
        <DoubleAnimation Duration="0:0:0.3" To="1"
          Storyboard.TargetProperty="(UIElement.Opacity)"
          Storyboard.TargetName="mouseoverRect"
          d:IsOptimized="True"/>
      </Storyboard>
    </VisualState>
    <VisualState x:Name="Pressed"/>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

A példa még mindig nem teljes értékű, ugyanis ha most rákattintunk a gombra, az a **Pressed** állapotba kerül, ami gyakorlatilag megegyezik a jelenlegi alapállapottal. A következőket szeretnénk: a gomb

Pressed állapotban „nyomódjon be” egy picit, illetve tartsa meg a már **MouseOver** állapotban összeszedett vizuális kinézetét. Az első nem jelent különösebb problémát, nincs más dolgunk, mint a **ControlTemplate**-ben található gyökérelemen elhelyezni egy **ScaleTransform** típusú **RenderTransform** objektumot, majd ezt animálni. A második probléma viszont ennél összetettebb. Vegyük észre, hogy nem mást akarunk, mint vizuális átmenethez kapcsolódóan egy megkötést alkalmazni! Azt akarjuk meghatározni, hogy **MouseOver** állapotból átmenve **Pressed** állapotba mi és hogyan történjen. Egészen pontosan a tranzíció átmenete itt 0 másodperc kell legyen, az **Opacity** értéke pedig 1 kell, hogy maradjon. Ezt, akárcsak az alapértelmezett átmeneteket, szintén **VisualTransition** objektumok segítségével érhetjük el. Ez utóbbi átmenethez tartozó kód az alábbi:

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="CommonStates">
    <VisualStateGroup.Transitions>
      <VisualTransition GeneratedDuration="0:0:0.3">
        <VisualTransition.GeneratedEasingFunction>
          <CubicEase EasingMode="EaseIn"/>
        </VisualTransition.GeneratedEasingFunction>
      </VisualTransition>
      <VisualTransition From="MouseOver" GeneratedDuration="0:0:0" To="Pressed">
        <Storyboard>
          <DoubleAnimation Duration="0" To="1"
            Storyboard.TargetProperty="(UIElement.Opacity)"
            Storyboard.TargetName="mouseOverRect"/>
        </Storyboard>
      </VisualTransition>
    </VisualStateGroup.Transitions>
    <VisualState x:Name="Normal"/>
    <VisualState x:Name="MouseOver">
      <Storyboard>
        <DoubleAnimation Duration="0:0:0.3" To="1"
          Storyboard.TargetProperty="(UIElement.Opacity)"
          Storyboard.TargetName="mouseOverRect"
          d:IsOptimized="True"/>
      </Storyboard>
    </VisualState>
    <VisualState x:Name="Pressed"/>
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

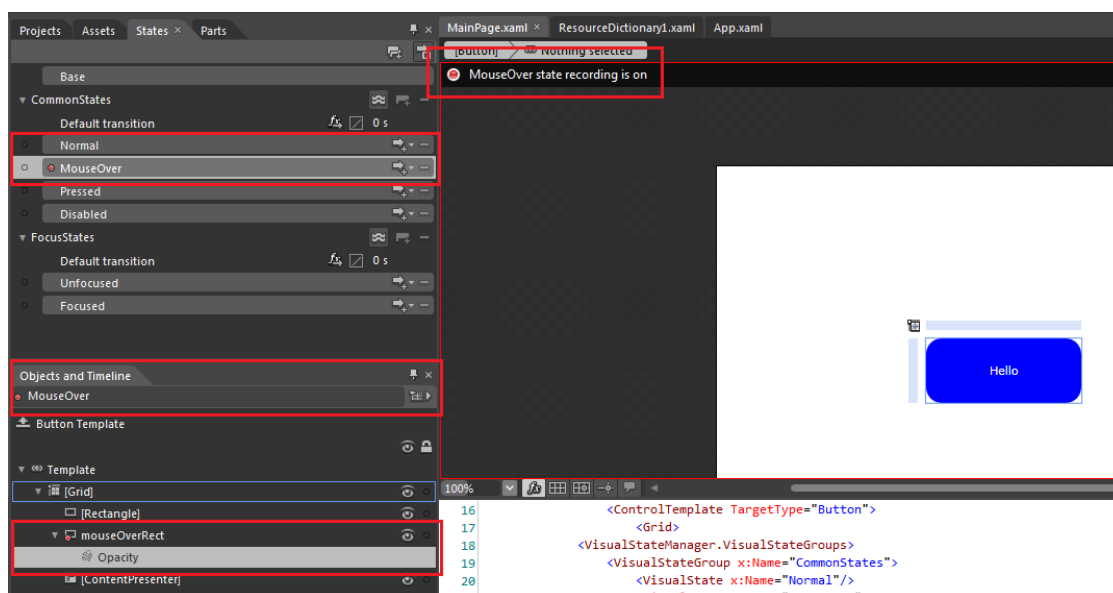
A fenti **VisualTransition** egyértelműen meghatározza, hogy a **MouseOver**-ből **Pressed** állapotba nulla másodperc alatt megyünk, és a kapcsolódó **Storyboard** pedig az **Opacity** értéket állítja 1-be, ezáltal azt szimulálva a felhasználónak, hogy valójában nem is történik változás.

Vizuális állapotokhoz tartozó megjelenés definiálására az Expression Blendből is van lehetőség.

A States panelen található vizuális állapotok segítségével minden állapothoz külön meghatározható a kinézet. Base állapot esetén a tulajdonságok állítgatása nincs hatással az átmenetekre. Azonban ha egy **VisualState** van kiválasztva, akkor a tervezőfelület körül egy piros keret jelenik meg, miszerint „... state recording is on”, azaz bármit is állítunk most be a Properties ablakban, az garantáltan egy animáció része lesz, ehhez a vizuális állapothoz kapcsolódóan (5-10 ábra).

Az ábrán jól látható, hogy a **MouseOver** vizuális állapotban vagyunk, *state recording* módban (azaz minden módosításunk rögzítésre kerül). Továbbá, egy animációt definiáltunk a **mouseOverRect** objektumhoz, amelynek egyébként az **Opacity** tulajdonságát animáljuk.

Figyelem: Ne felejtjük el kikapcsolni a state recording-ot, ha nem egy állapotváltáshoz kapcsolódó tulajdonságot állítunk be!



5-10 ábra: Vizuális állapotok szerkesztése az Expression Blendben

A vizuális állapotok rendkívül hatékony megoldást biztosítanak a felhasználói felület különböző állapotainak kezelésére. Népszerűségét mi sem bizonyítja jobban, mint az, hogy a Windows Presentation Foundation is átvette idővel ezt a megoldást, korábban ez csak Silverlightben volt elérhető.

A **VisualState**-ek nem csak **ControlTemplate**-ben használhatók, hanem szinte bárhol, például a főoldalon egy panel láthatóságát (ki-be csúsztatását) biztosíthatja. Egy vezérlő vizuális állapotát kódból is meg lehet változtatni, a **VisualStateManager.GotoState()** hívása segítségével. További információk a CustomControlok készítése fejezetben találhatók erről a témáról.

Komplexebb vezérlők testreszabása

A **Button** talán a legegyszerűbb vezérlő, ami létezik. De mi a helyzet komplexebb vezérlőkkel, mint például egy **ScrollBar**, **Slider** vagy **ListBox**? Ezeknek a vezérlőknek a sablonjában jóval bonyolultabb vizuális fa található, és a fa egyes részeihez funkcionalitás is tartozik, mint például görgetés, a **Slider** értékének változtatása, stb. Ezek megértéséhez egy új fogalom, az ún. *Template Part* ismeretére van szükség. Ezeknek a komplex vezérlőknek a testreszabásához és a Template Partok működésének megértéséhez meg kell ismerkedni a vezérlők pontos működésével. Erről további részleteket a 7. fejezetben olvashatunk.

Összefoglalás

Az egységes témák és kinézet kialakítása fontos minden alkalmazástípus esetén. A Silverlight komoly eszköztárat biztosít számunkra a stílusok és az erőforrások formájában, melyek segítségével a hordozhatóság és az újrafelhasználhatóság megteremtése jelentősen leegyszerűsödhet. A vezérlők minden részletére kiterjedő teljes testreszabását a **ControlTemplate**-ek támogatják, eddig soha nem látott mértékben. Egy-egy **Template** testreszabása során vizuális állapotok és ún. **TemplateBinding**-ok segítenek bennünket, hogy számos - korábban a kinézethez kapcsolódó funkcionális kódot igénylő - feladatot most deklaratív módon, XAML segítségével, tervezőeszközöket használva oldjunk meg. Elmondhatjuk, hogy egy bonyolultabb vezérlő testreszabásának elengedhetetlen eszköze az Expression Blend 4, amely kiemelkedő produktivitást biztosít használója számára.

6. Adatok kezelése

Ebben a fejezetben az adatkezelés *adatmegjelenítési* részével fogunk foglalkozni. (Az adatkezelés egy másik kulcsfontosságú komponenséről, az *adatelérésről* a 8. fejezetben olvashatunk bővebben.) Itt a Silverlight nyújtotta lehetőségeket tekintjük át az egyszerű objektumok reprezentációjától a nagyobb méretű adathalmazok vizualizációjáig. Kalandunk során megnézzük, hogy milyen sokféleképpen alakíthatjuk át az adatainkat olyan formájúvá, hogy az a felhasználói felületen már érthető és értékes információként jelenjen meg. Mindemellett utánajárunk annak is, milyen módon tudjuk elősegíteni az üzleti alkalmazásunk gördülékeny használatát.

Hogyan használjuk az adatokat?

Különböző adatforrások, különböző adatmennyiségek

Bár ez a fejezet nem foglalkozik explicit módon az adatelérés kérdésével – hanem feltételezzük, hogy a megfelelő adathalmaz rendelkezésünkre áll – ,ettől függetlenül érdemes végiggondolni, hogy milyen adatokkal is dolgozunk. Pontosabban fogalmazva: mennyi adatról van szó, és azok honnan érkeznek?

Az adatok nagyon sokféle helyről származhatnak, kezdve a felhasználótól egészen egy távoli adatbázis szerverig szinte bárhonnan érkezhettek. Ebben a fejezetben csak a back-end-től kapott, kliensoldalon CLR-objektumként, listaként, esetleg egyéb formátumban ábrázolt adatokkal foglalkozunk. (A 10. fejezet adat validációs részében olvashatunk bővebben a kliensoldalról érkező adatok kezeléséről.) Persze az a kérdés is érdekes lehet, hogy mi a teendő akkor, ha a kinézet tervezésekor szükségünk lenne néhány mintaadatra a felhasználói felület kipróbálásához, viszont az adatok elérése erőforrásigényes és hosszadalmas. Ilyenkor jönnek a képbe a *mintaadatok*, amelyekkel részletesen a *Listás adatmegjelenítés* című alfejezetben fogunk foglalkozni.

Az adatrepresentációt befolyásoló másik fontos tényező az adatmennyiség. Könnyen belátható, hogy egyetlen objektum megjelenítése és egy adathalmaz vizualizációja teljesen más szemléletet kíván. Vagy mégsem? Silverlight használatánál ez a két dolog nem is esik olyan messze egymástól, ugyanis egy adathalmaz reprezentálása olyan módon történik, hogy megmondjuk, miként nézzen ki egy adott elem, és ezt a megjelenítési sémát fogja alkalmazni a rendszer a többi elemre is. Persze, az egyes elemek egymáshoz viszonyított elhelyezkedését, helyzetét is meg kell adnunk, de ez már egy másik kérdés.

Adatmegjelenítési lehetőségek

Induljunk ki a legegyszerűbb esetből, amikor csak egyetlen objektumunk van, és ezt akarjuk megjeleníteni egy tetszetős felhasználói felületen! Tegyük fel, hogy ebből a fent említett osztályból tulajdonságokon keresztül kinyerhetők a megjelenítendő adatok! Ezeket az értékeket összekapcsolhatjuk bármely vezérlő bármely tulajdonságával, feltéve hogy az adattípusuk megegyezik. Persze ahhoz, hogy a vezérlő meg is jelenítse őket, arra is szükségünk van, hogy az információt tároló objektumot beállítsuk a vezérlő *adatforrásának*. Ha ezzel megvagyunk, akkor a többi már a Silverlight feladata. Azt a folyamatot, amely ezt az egészet kezeli, *adatkötésnek* nevezzük.

Abban az esetben, ha adatok egy gyűjteménye áll rendelkezésünkre, már eléggé leszűkül a felhasználható vezérlők készlete. Ekkor listás vagy táblázatos adatmegjelenítésre képes vezérlőket kell használnunk, pontosabban olyanokat, amelyek az **ItemsControl** osztályból származnak. Listás megjelenítő választása esetén meg kell adnunk egy sablont (**ItemTemplate**-t), amely definiálja egy elem kinézetét, majd ezt ismételi az adathalmaz összes többi elemére. Ezzel egyrészt definiáljuk az adott elemet reprezentáló vezérlők elhelyezkedését, másrészt pedig „adatkötjük” őket a megfelelő tulajdonságokhoz.

Amennyiben a táblázatos megjelenítő mellett tesszük le a voksunkat, nincs szükségünk arra, hogy definiáljunk **ItemTemplate**-et, ugyanis az adathalmaz egyes elemei egymás alatt fognak elhelyezkedni,

míg a tulajdonságaik (a rekordok mezői) pedig egymás mellett — táblázatos formában. A kinézet testreszabhatósága emiatt meglehetősen korlátozott a listás megjelenítéshez képest, ugyanis egyedül a mezők típusán (*szöveg, logikai érték, gomb, egyedi felület*) lehet változtatni. Ebben a fejezetben csak a listás adatmegjelenítéssel foglalkozunk, a táblázatosat a 11. fejezet tárgyalja.

Bár a Silverlightos vezérlőket tartalmazó arzenálban alaptól nem kapott helyet a Diagram (**Chart** vezérlő), ettől függetlenül van arra is lehetőségünk, hogy ezt használjuk az adataink vizualizációjához. Ehhez a *Silverlight Toolkit*re lesz szükségünk, melyben több különböző diagramtípus közül is válogathatunk. Sőt, olyan vezérlő is elérhető az interneten (**PivotViewer**), mely segítségével hatalmas mennyiségű adatot tudunk képek és metainformációk kombinációjaként úgy kipublikálni, hogy az könnyen kezelhető, szűrhető, kereshető és rendezhető legyen.

Az előző bekezdésben említett két vezérlővel ebben a fejezetben nem foglalkozunk, viszont érdemes lehet utána nézni, milyen új kapuk nyílnak meg előttünk, ha ezeket használjuk. Kiindulási alapként az alábbi két linket ajánlom a figyelmetekbe:

- <http://silverlight.codeplex.com/wikipage?title=Silverlight%20Toolkit%20Overview%20Part%202>
- <http://www.microsoft.com/silverlight/pivotviewer/>

A megjelenítési réteg és az adatréteg összekapcsolási módjai

Amikor adatmegjelenítésről beszélünk, akkor mindig úgy gondolunk a két réteg közötti kommunikációra, hogy az adatréteg a szolgáltató, a megjelenítési réteg pedig a fogyasztó. Silverlight esetén ez a két szerepkör felcserélhető, vagyis a felhasználói felületen keresztül is érkezhetsz új információ, amit a háttérben lévő üzleti objektumnak automatikusan átadhatunk egyetlen sornyi C# kód nélkül, szintiszta deklaratív módon (*automatikus változáskövetés*). Nézzünk erre egy egyszerű példát: Tegyük fel, hogy van egy **TextBox** vezérlőnk, mely egy adatforrásból nyeri a tartalmát, de ha a felhasználó módosítja azt, akkor az új érték egyből visszairódik az adatforrásba. Az eredeti felállást egyirányú adatkötésnek nevezzük, míg azt, ahol a célobjektum változása esetén az adat a forrásobjektumba is visszakerül, kétirányúnak.

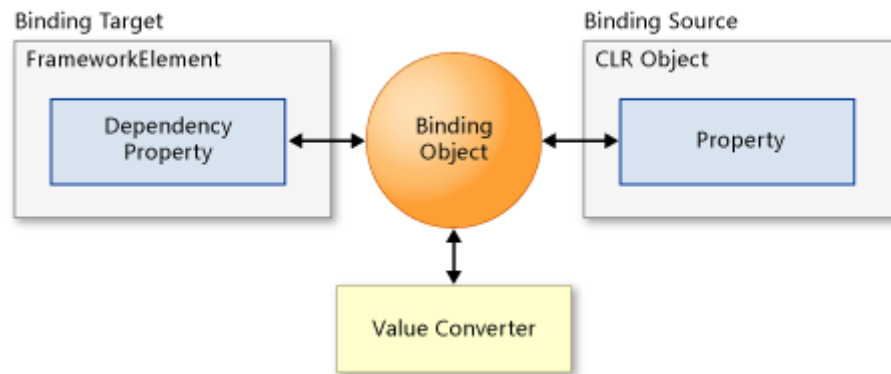
A kétirányú adatkötés rendkívül hasznos és fontos eszköz az üzleti alkalmazásoknál. A fontosságát mi sem bizonyítja jobban, mint hogy egy külön architektúrális minta is született az elmúlt években, mely erőteljesen épít erre a szolgáltatásra. Ez a *Model-View-ViewModel*, amellyel a 10. fejezetben foglalkozunk bővebben. Ugyanebben a fejezetben kitérünk arra is, hogy miként lehetséges a felhasználtól érkező adatok automatikus ellenőrzése (pl. a hossza megfelelő-e, egy adott tartományon belül található az érték, stb.).

Az adatkötés alapjai

Az adatkötés folyamata

Mint ahogyan azt az előző részben is már említettem, az adatkötés az a folyamat, mely a felhasználói felület és az adatobjektum közötti kapcsolatot létrehozza, és az adatáramlást felügyeli, kezeli. Ennek egyik nagy előnye, hogy így extra kódolás nélkül konzisztensen tudjuk tartani mind a felületet, mind pedig az üzleti objektumunkat.

A Silverlightban az adatkötés vezérlők tulajdonságai és objektumok tulajdonságai között létesíthető. Az adatkötés alapkoncepcióját a 6-1 ábra jól szemlélteti.



6-1 ábra: Az adatkötés szerkezete

(Létezik egy speciális eset is, amikor a képlet jobb oldala is egy vezérlő, de ezzel majd csak a *Vezérlők közötti adatkötés* című alfejezetben fogunk foglalkozni.)

A 6-1 ábrán jól látható, hogy bármilyen CLR objektum megjeleníthető adatkötés segítségével, feltéve, hogy a benne tárolt információk tulajdonságokon keresztül elérhetőek. Persze, nem minden esetben ugyanabban a formátumban tároljuk el az adatokat az üzleti objektumban, mint ahogyan azt a felhasználói felületen megjeleníteni szeretnénk (vegyük például a színeket: tárolásuk rgb formában — 3db 0 és 255 közötti érték —, megjelenítésük ecset segítségével #aarrggbb hexadecimális formában). Így kerül a képbe a **ValueConverter**, amely az adatforrásból vett adatot átalakítja a felület számára is értelmezhető formájúvá. Ezzel a *Konverterek használata* című alfejezetben foglalkozunk.

DependencyProperty és a Silverlight Tulajdonság rendszere

A 6-1 ábra bal oldalán érdekes dolog látható. Az ábra szerint bármilyen **FrameworkElement** (a **Control** és a **Panel** osztály ősszánya) lehet adatmegjelenítésre alkalmas objektum, ha van úgynevezett *DependencyProperty*-je.

Röviden a *DependencyProperty* (a továbbiakban csak DP) egy olyan speciális tulajdonság, amelynek az értéke más objektumtól érkező adatok alapján kerül dinamikusan kiszámításra, tehát az értéke függhet más objektum értékétől, mint például: *Style*, *Resource* vagy akár egy folyamattól, például: *animáció*, *adatkötés*. A rendszer prioritás alapú. Hogy ez mit is jelent, azt egy egyszerű példán keresztül szemléltetem. Tegyük fel, hogy egy vezérlő valamely tulajdonságának egy stíusból beállítunk egy 5-ös értéket! Ezenkívül egy lokális értékadással 7-re növeljük a tulajdonság értékét. Majd pedig egy adott esemény bekövetkezésekor az értéket 3-tól 10-ig animáljuk. A prioritásos sor miatt (amelyben az elemek az alábbi sorrendben szerepelnek: animáció > explicit értékadás > stílus), a vezérlő betöltése után a tulajdonság értéke 7 lesz, az animáció alatt pedig 3 és 10 közötti értékek valamelyike. A teljes precedencia lista az alábbi címen érhető el: [http://msdn.microsoft.com/en-us/library/cc265148\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc265148(VS.95).aspx).

Az, hogy egy tulajdonság *DependencyProperty*, azzal jár együtt, hogy jó néhány olyan funkcionalitással alaphoz rendelkezik a fentiekén kívül, mint például a változásértesítés (ha megváltozik az értéke, akkor képes értesíteni erről például a felhasználói felületet), alapérték megadás (a tulajdonság alapértelmezett értékének beállíthatósága), stb. Ezen szolgáltatások összességét szokás *Silverlight Property System*-nek nevezni.

Ahhoz, hogy mindez működhessen, egy speciális tárolóra van szükségünk, amely eltárolja és felügyeli a tulajdonságba elhelyezett értéket. Ez a tároló a 6-1 ábrán lévő *Binding Target*, mely kötelezően a **DependencyObject** osztályból származik.

Egy DP tulajdonság definiálását mutatja be a következő példa:

```
public partial class MainPage : UserControl
{
    public static readonly DependencyProperty IsThisGoodProperty =
        DependencyProperty.Register( "IsThisGood", typeof(Boolean), typeof(this), null);
    public bool IsThisGood
    {
        get { return (bool)base.GetValue(IsThisGoodProperty); }
        set { base.SetValue(IsThisGoodProperty, value); }
    }
}
```

A kód két részből áll, egy regisztrációból és egy csomagoló tulajdonságból. A regisztráló függvény (**DependencyProperty.Register**) négy paramétert vár. Az első a DP azonosítója (neve), a második a tulajdonság típusa, a harmadik az őt tároló **DependencyObject** típusa (ez lehet például egy **UserControl** osztály is). Az utolsó paramétere pedig egy **PropertyMetadata** példány, amiben megadható egy a változás bekövetkezése esetén meghívandó *callback függvény*, illetve a tulajdonság alapértéke. A **Register** függvény által visszaadott értéket eltároljuk egy **DependencyProperty** típusú statikus, csak olvasható adattagban. A DP értékét lekérdezni és beállítani a **GetValue**, illetve **SetValue** függvényekkel lehet. Azért, hogy a DP-t úgy lehessen kívülről használni, mint bármely más CLR tulajdonságot, egy csomagoló tulajdonságba szokás beágyazni a függvényhívásokat.

Silverlight esetén a vezérlők szinte összes tulajdonsága ilyen módon van megvalósítva, ezért majdnem minden tulajdonságuk adatköthető. Ha szeretnénk megbizonyosodni arról, hogy az adott tulajdonság valóban DP-e, akkor azt kell megnéznünk, hogy a vezérlőnek van-e olyan adattagja, melynek a neve {a_vizsgált_tulajdonságneve} + „Property” alakú. Saját DP-re általában egyedi vezérlők fejlesztésénél lehet szükségünk, lásd következő fejezet.

Adatkötés alapjai egy egyszerű példán keresztül

Az eddig tanultakat most nézzük meg a gyakorlatban is! A példánk három fő lépésből fog állni: adatobjektum definiálása, adatforrás beállítása, végül pedig a tulajdonságok összekapcsolása.

Hozzunk létre egy új Silverlight Application típusú alkalmazást *DataBinding_Demo* néven!

1. Adjunk egy új osztályt a projekthez **Dolgozo.cs** néven! Ez az osztály néhány tulajdonságot definiál:

```
public class Dolgozo
{
    public string Vezeteknev { get; set; }
    public string Keresztnev { get; set; }
    public int Eletkor { get; set; }
    public DateTime Csatlakozott { get; set; }
}
```

2. Nyissuk meg a **MainPage.xaml.cs** fájlt, majd a konstruktorban iratkozzunk fel a **Loaded** eseményére! Ezen belül hozzunk létre egy példányt az előbb elkészített **Dolgozo** osztályból! Végül ezt állítsuk be az oldal **DataContext** tulajdonságának! Íme, ennek a lépésnek a kódja:

```
public partial class MainPage : UserControl
{
    public MainPage()
    {
        Loaded += (s, e) =>
        {
            Dolgozo d = new Dolgozo() { Vezeteknev = "Dotnet", Keresztnev = "Elek",
                Eletkor = 25, Csatlakozott = DateTime.Now };
            this.DataContext = d;
        };
    }
}
```

A **DataContext** tulajdonság segítségével tudjuk megadni a vezérlők adatforrását, vagyis adatkötéskor innen kérjük le az adatokat. A dolog szépsége, hogy az adatforrásként beállított objektum képes lefelé csorogni, vagyis elég csak egyszer, a konténer vezérlőnél beállítani ezt a tulajdonságot, utána az üzleti objektumot az összes gyerekvezérlő eléri, megőröklí.

3. Most váltsunk át a **MainPage.xaml** fájlra, és a **LayoutRoot** vezérlőn belül definiáljunk 2 oszlopot és 4 sort! Az első oszlopba **TextBlock** vezérlőket pakoljunk, a másodikba pedig **TextBox**-okat! A **TextBlock** vezérlők címkeként funkcionálnak majd, melyek **Text** tulajdonsága rendre a következő: „Vezetéknév:”, „Keresztnév:”, „Életkor:”, „Csatlakozás dátuma:”. Az XAML kódunk jelenleg az alábbi módon néz ki:

```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="Auto" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>

  <TextBlock Text="Vezetéknév:" Grid.Row="0" />
  <TextBlock Text="Keresztnév:" Grid.Row="1" />
  <TextBlock Text="Életkor:" Grid.Row="2" />
  <TextBlock Text="Csatlakozás dátuma:" Grid.Row="3" />

  <TextBox Grid.Row="0" Grid.Column="1" />
  <TextBox Grid.Row="1" Grid.Column="1" />
  <TextBox Grid.Row="2" Grid.Column="1" />
  <TextBox Grid.Row="3" Grid.Column="1" />
</Grid>
```

Az adatkötést egy speciális *markup extension* (a jelölést leíró nyelvi kiterjesztés) segítségével tudjuk megvalósítani, hasonlóan, mint ahogy az erőforrásokra hivatkozunk: kapcsos zárójelek közé kell tennünk egy speciális kulcsszót, majd pedig egy kulcsértéket. Jelen esetben a kulcsszó a **Binding**, a kulcsérték pedig az adatforrás objektum megjelenítendő/adatkötendő tulajdonságának a neve, például **{Binding Eletkor}**. Ezek után a TextBoxok XAML kódja az alábbi módon fog kinézni:

```
...
<TextBlock Text="Csatlakozás dátuma:" Grid.Row="3" />

<TextBox Grid.Row="0" Grid.Column="1" Text="{Binding Vezeteknev}" />
<TextBox Grid.Row="1" Grid.Column="1" Text="{Binding Keresztnev}" />
<TextBox Grid.Row="2" Grid.Column="1" Text="{Binding Eletkor}" />
<TextBox Grid.Row="3" Grid.Column="1" Text="{Binding Csatlakozott}"/>
...
```

Az adatkötésnek köszönhetően az alkalmazás a **TextBox** vezérlőkben megjeleníti a **DataContext**-nél beállított **Dolgozo** példány tulajdonságait, mint ahogy azt a 6-2 ábra is szemlélteti.

Vezetéknév:	Dotnet
Keresztnév:	Elek
Életkor:	25
Csatlakozás dátuma:	10/14/2010 11:39:47 AM

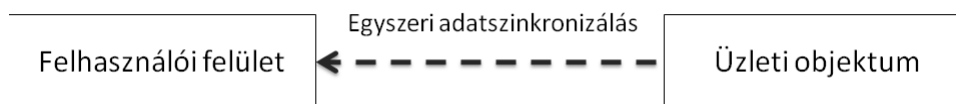
6-2 ábra: Adatkötés eredménye futásidőben

Adatkötési lehetőségek, testreszabás

Adatkötési módok

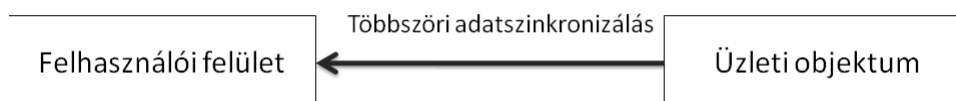
Az adatkötésnek azon kívül, hogy kényelmesebb, mint kódból egyesével összekapcsolni a tulajdonságokat, van még néhány előnye. Ezek közül az egyik legfontosabb a változáskezelési képessége, amely kapcsán három adatkötési típusról beszélhetünk: *OneTime*, *OneWay*, *TwoWay*.

A **OneTime** típusú adatkötés a legegyszerűbb mind közül. Itt nincs változáskövetés, és ahogyan a neve is utal rá, csak egyszer történik adatszinkronizáció (lásd 6-3 ábra). Ez akkor valósul meg, amikor a felület betöltődik, és lekérdezi az adatforrásából a megfelelő mezőértékeket. Ezután, ha az adatforrás (üzleti objektum) módosul, a felhasználói felület nem fog automatikusan frissülni, illetve az adatok szerkesztésekor sem fog egyből az új érték a háttérben lévő objektumba visszaíródni.



6-3 ábra – OneTime módú adatkötés

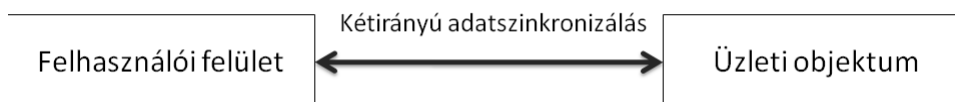
A **OneWay** annyival tud többet **OneTime** társánál, hogy itt az objektumtól a felület felé irányú változáskövetés megoldott (lásd 6-4 ábra). Ez azért nagyon hasznos, mivel így ha módosul az adatforrás, akkor nincs szükség egyetlen sor kódra se a felület frissítéséhez, hanem a megjelenítő rendszer képes saját magát automatikusan konzisztens állapotba hozni. Mellesleg ez az alapértelmezett viselkedés adatkötés esetén.



6-4 ábra – OneWay módú adatkötés

A **TwoWay** mód a **OneWay** funkcionalitását annyiban bővíti ki, hogy a másik irányú (a felülettől az objektum felé történő) automatikus frissítési lehetőséget is támogatja (lásd 6-5 ábra). Ez hatalmas előnyt jelent, ha engedélyezni szeretnénk a felhasználónak az adatobjektumaink módosíthatóságát, és minimalizálni akarjuk a szerkesztéshez és frissítéshez szükséges kódot.

Fontos: Itt az adatobjektum alatt ne csak az adatbázisokból (vagy egyéb forrásból) kinyert entitásokra gondoljunk, hanem például csúszkák állapotának vagy egér pozíciójának tárolására is! Vagyis nemcsak szöveges inputban gondolkodhatunk, hanem bármilyen bemenő jel érzékelésében, feldolgozásában és eltárolásában. Persze, amikor a felhasználótól érkezik adat, akkor minden esetben érdemes (sőt kifejezetten ajánlott) azt megvizsgálni, hogy megfelel-e bizonyos kritériumnak (esetleg nem rosszindulatú-e). Ezzel a témakörrel a 10. fejezetben foglalkozunk részletesen.



6-5 ábra – TwoWay módú adatkötés

Ahhoz, hogy az utóbbi kettő, vagyis az egy-, illetve kétirányú adatkötés működjön, az adatobjektumoknak meg kell valósítaniuk az **INotifyPropertyChanged** interfészt.

A kétirányú adatkötés és az INotifyPropetyChanged kapcsolata

Először is tekintsük meg az **INotifyPropertyChanged** interfész definícióját, amely a **System.ComponentModel** névtéren belül található:

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

Látható, hogy egyetlen adattagja van csak, ami egy esemény. Ezt az eseményt akkor kell meghívni, ha az adatobjektum valamely kívülről is elérhető (pontosabban adatkötésben résztvevő) tulajdonsága megváltozik. Vagyis explicit módon meg kell mondanunk, hogy most történt valami, amire a *Binding Engine*-nek figyelnie kell, és ezáltal értesíteni kell a felületet, hogy kérje le ismét az adatokat és frissítse a vezérlőket.

Silverlight, illetve WPF esetén az interfész implementálása általában oly módon történik, hogy egy segédmetódust vezetünk be, és azon keresztül hívjuk meg ezt az eseményt. Ennek a kódja általában az alábbi módon szokott kinézni:

```
private void RaisePropertyChanged(string propName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propName));
}
```

Az interfésznek persze sokféle egyéb implementálása is lehetséges, mint ahogyan azt a következő oldal is taglalja: <http://compositeextensions.codeplex.com/Thread/View.aspx?ThreadId=53731>. Ajánlom mindenkinek, hogy egyszer fussa át az oldalt, ha másért nem, akkor érdekesség gyanánt.

Az interfész implementálása után a következő lépés a **RaisePropertyChanged** segédfüggvény meghívása a megfelelő helyen. Ez a hely nem más, mint a tulajdonságok *set* metódusa. Itt egyből belefutunk abba a problémába, hogy az előző példaalkalmazásnál mi *auto-implemented property*-ket használtunk, vagyis a **set** és a **get** kódjai dinamikusan generáltak, a privát adattagok pedig implicit módon vannak jelen. Itt sajnos a régi, jól bevált tulajdonságokra van szükségünk, nincs lehetőség a C# 3.0-ban debütált újdonság használatára. A **Vezeteknev** tulajdonságon keresztül mutatom meg, miként kell átírni az automatikusan

implementált tulajdonságokat, hogy azok jelezzék a Silverlight felé, ha módosultak. Íme, a módosított **Vezeteknev**:

```
private string _vezeteknev;
public string Vezeteknev
{
    get { return _vezeteknev; }
    set
    {
        _vezeteknev = value;
        RaisePropertyChanged("Vezeteknev");
    }
}
```

Végezetül pedig, valahogy így kell kinéznie a **Dolgozo.cs** fájlunk átdolgozott kódjának:

```
public class Dolgozo: INotifyPropertyChanged
{
    private string _vezeteknev;
    public string Vezeteknev
    {
        get { return _vezeteknev; }
        set
        {
            _vezeteknev = value;
            RaisePropertyChanged("Vezeteknev");
        }
    }

    private string _keresztnev;
    ...

    public event PropertyChangedEventHandler PropertyChanged;
    private void RaisePropertyChanged(string propName)
    { ... }
}
```

Ezek után nézzük meg, mit kell tennünk ahhoz az XAML kódban, hogy az adatforrás változásáról kapjunk értesítést. Itt szerencsére nem kell sokat gépelnünk, mindössze a **{Binding propName}**-eket kell kibővítenünk a **Mode=OneWay** | **TwoWay** valamelyikével. Válasszuk most a kétirányú adatkötést, és ahhoz, hogy lássuk, valóban módosul-e a háttérben az adatforrás, másoljuk le a **Grid** teljes kódját és a két **Grid** köré tegyünk egy **StackPanel**-t (ez legyen az új **LayoutRoot**)! (A két **Grid**-nek azonos lesz az adatforrása, ezért ha az egyiknél módosítjuk valamelyik tulajdonság értékét, akkor az a másikon is módosulni fog.) Íme, a módosított **MainPage.xaml** kódja:

```
<StackPanel x:Name="LayoutRoot" Background="White">
    <Grid>
        ...
        <TextBlock Text="Csatlakozás dátuma:" Grid.Row="3" />
        <TextBox Grid.Row="0" Grid.Column="1" Text="{Binding Vezeteknev, Mode=TwoWay}"/>
        <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding Keresztnev, Mode=TwoWay}"/>
        <TextBox Grid.Row="2" Grid.Column="1" Text="{Binding Eletkor, Mode=TwoWay}"/>
        <TextBox Grid.Row="3" Grid.Column="1" Text="{Binding Csatlakozott, Mode=TwoWay}"/>
    </Grid>
    <Grid>
        <!-- Ennek a Grid-nek is ugyanaz a kódja, mint az előzőnek -->
        ...
        <TextBlock Text="Csatlakozás dátuma:" Grid.Row="3" />
        <TextBox Grid.Row="0" Grid.Column="1" Text="{Binding Vezeteknev, Mode=TwoWay}"/>
        <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding Keresztnev, Mode=TwoWay}"/>
        <TextBox Grid.Row="2" Grid.Column="1" Text="{Binding Eletkor, Mode=TwoWay}"/>
    </Grid>
</StackPanel>
```

```
<TextBox Grid.Row="3" Grid.Column="1" Text="{Binding Csatlakozott, Mode=TwoWay}"/>
</Grid>
</StackPanel>
```

A kétirányú adatkötést tartalmazó példa működését a 6-6 és 6-7 ábra szemlélteti.

Vezetéknév:	DotNet
Keresztnév:	Elek
Életkor:	25
Csatlakozás dátuma:	10/14/2010 2:30:40 PM
Vezetéknév:	Dotnet
Keresztnév:	Elek
Életkor:	25
Csatlakozás dátuma:	10/14/2010 2:30:40 PM

6-6 ábra: A felsőben nagy N-esre módosítom a Dotnetet

Vezetéknév:	DotNet
Keresztnév:	Elek
Életkor:	25
Csatlakozás dátuma:	10/14/2010 2:30:40 PM
Vezetéknév:	DotNet
Keresztnév:	Elek
Életkor:	25
Csatlakozás dátuma:	10/14/2010 2:30:40 PM

6-7 ábra: A fókusz elvesztése után nagy N-es lesz a lenti is

Említés szintjén néhány szót ejtenék arról a lehetőségről is, hogy nem minden esetben tökéletes megoldás az automatikus adatmentés, például összetett (több tulajdonságot érintő) adat érvényességének ellenőrzésekor. Ilyenkor megmondhatjuk a Silverlightnak, hogy mi szeretnénk explicit módon megadni, hogy mikor is történjen az adatmentés a háttérben lévő adatforrásba. Erről bővebb információt és egy egyszerű példát az alábbi címen találhattok: [http://msdn.microsoft.com/en-us/library/cc278072\(VS.95\).aspx#updating_the_data_source](http://msdn.microsoft.com/en-us/library/cc278072(VS.95).aspx#updating_the_data_source).

Megjelenített adat formázása, esetleges hiányosságok, hibák kezelése

Ha jobban megnézzük a 6-6 vagy a 6-7 ábrát, akkor észrevehetünk egy apró kis szépséghibát a dátum megjelenítésében: azok nem a Magyarországon használt formátumban vannak. A Silverlight korábbi verzióiban ahhoz, hogy a dátumokat a megfelelő formában jelenítsük meg, egy saját konvertert kellett írunk. Szerencsére a Silverlight 4 erre egy egyszerűbb megoldást nyújt, a **StringFormat** paraméter képében. Itt is hasonlóan az ASP.NET-es adatkötéshez vagy a **String** osztály **Format** függvényéhez van egy *placeholder* (helyőrző) és egy *format pattern* (formázó minta). A helyőrző helyére fog bekerülni az adott érték a formázó mintának megfelelően. Íme, a dátum helyes magyar formában történő megjelenítéséhez szükséges kód:

```
<TextBox Grid.Row="3" Grid.Column="1" Text="{Binding Csatlakozott, Mode=TwoWay,
StringFormat=yyyy.MM.dd}"/>
```


Az **y** az évet, az **M** a hónapot, a **d** pedig a napot jelöli, így a végeredmény valami hasonló lesz: „2010.10.14”. Ennél a **StringFormat**-os példánál csak a mintát adtuk meg, a helyőrzőt nem.

Most nézzünk egy kicsit bonyolultabb példát, ahol helyőrzőt is használunk! Tegyük fel, hogy van egy olyan tulajdonságunk, amelyben egy adott pénznembeli értéket tárolunk el (ez általában **decimal** vagy **double** típusú), és elé ki akarjuk írni, hogy „Bevétel:”, de ehhez nem szeretnénk egy külön **TextBlock** vezérlőt használni! Ennek a megvalósítása az alábbi módon történik:

```
<TextBlock Text="{Binding Bevetel, StringFormat='Bevétel: \{0:C\}'}" />
```

Ebben az esetben a **StringFormat** értékét aposztrófok (') közé írjuk, így tetszőleges szöveget el tudunk helyezni az adatkötött érték köré. A helyőrző itt a {0}, a formázó minta pedig a C (mint *currency*, ami a pénznem formátumát az operációs rendszer beállításai közül veszi), és azért van a nyitó és csukó kapcsos jelek előtt egy-egy fordított perjel, hogy az XAML Parser helyesen értelmezze őket. Az adatkötés eredményét a 6-8 ábra mutatja.

Bevétel: \$10.00

6-8 ábra: Pénznem formázó minta használata

A Silverlight 4-ben a **StringFormat**-on kívül még két új **Binding** tulajdonságot vezettek be a részletesebb tesztreszabhatóság érdekében, ezek a **TargetValueNull** és a **FallbackValue**. A **TargetValueNull** arra való, hogy null érték esetén az általunk megadott értéket használja a vezérlő az adatkötéshez a null érték helyettesítésére, pl.: „Nem tartozik ehhez a kategóriához termék”. Ez kifejezetten hasznos lehet olyan relációs adatbázisoknál, ahol több mező is tartalmazhat null értéket. A **FallbackValue**-t arra használhatjuk, hogy értesítsük a felhasználót arról, ha a megadott adatforrásból nem sikerült kinyerni adatot (pl. kivétel vagy konvertálási hiba miatt), tehát valamilyen okból kifolyólag nem áll rendelkezésünkre a megfelelő információ. Íme, egy nagyon egyszerű példa e tulajdonságok használatára:

```
<TextBox Text="{Binding Csatlakozott, TargetNullValue='Ismeretlen'}"/>
<TextBox Text="{Binding Eletkor, FallbackValue='Adat jelenleg nem elérhető'}"/>
```

Konverterek használata

A konverterek létjogosultsága cseppet sem halványult a **StringFormat** paraméter bevezetése óta, hiszen ezzel bármilyen típust bármilyen más típusra konvertálhatunk át. (Például string → Kép, string → Ecset, int (tartomány alapján) → Rectangle, Enum → *).

A konverterek használata az alábbi 3 lépésből áll:

1. konverter létrehozása
2. konverterből statikus erőforrásként példány létrehozása
3. adatkötésnél konverter megadása, beállítása

A konverterek használatának bemutatásához egy előre elkészített projektből fogunk kiindulni, amely a könyvhöz tartozó forráskódok között található az alábbi mappában:

Ch6/03_Konverterek_hasznalata_Indulo.

Ebben a példában az eredeti **Dolgozo** osztályt fogjuk használni, amelyet kibővítünk néhány új tulajdonsággal: eltávolítjuk az adott dolgozó nemét, beosztását és telefonszámát is. Ezekkel a változtatásokkal az osztály új definíciója az alábbi:

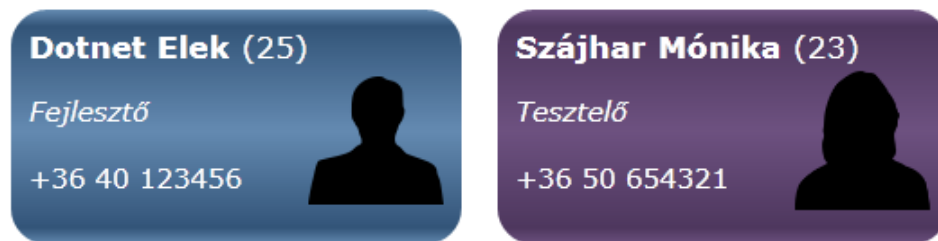

```

public enum nem { Ferfi, No };
public enum beosztas { Vezeto, Mernok, Fejleszto, Tesztelo };

public class Dolgozo
{
    public string Vezeteknev { get; set; }
    public string Keresztnev { get; set; }
    public nem Nem { get; set; }
    public int Eletkor { get; set; }
    public beosztas Beosztas { get; set; }
    public string Telefonszam { get; set; }
    public DateTime Csatlakozott { get; set; }
}

```

Az adatokat a 6-9 ábrán látható névjegykártyaszerű formában szeretnénk megjeleníteni.



6-9 ábra: Dolgozó névjegykártya (férfi/női)

A 6-9 ábrán látható két névjegykártya a háttérszínben, illetőleg a piktogramban tér el egymástól, melyek a **Nem** tulajdonságtól függenek (más-más típusú vizualizáció). Én most csak a piktogramhoz tartozó konverter elkészítését és használatát fogom lépésről lépésre bemutatni, a háttérhez tartozó konverter forráskódja a **HatterKonverter.cs** fájlban található.

A konverterek olyan osztályok, melyek megvalósítják az **IValueConverter** nevezetű interfészt, mely a **System.Windows.Data** névtérben található. Ez az interface két metódust definiál: a **Convert** és a **ConvertBack** metódusokat. Mint ahogyan az a nevükből is kitalálható, az első az adatobjektumbeli értéket alakítja át UI számára is értelmezhetővé, míg utóbbi a felületről érkező információt transzformálja a tároláshoz szükséges formátumra. Mindkettő visszatérési értéke **object**, így tehát bármilyen típusra konvertálhatunk. A paraméterlistájuk is megegyezik, amely az alábbi:

- **object value**: ez tartalmazza a konvertálandó értéket.
- **Type targetType**: **Convert**-nél az adott vezérlő adatkötött tulajdonságának típusa, **ConvertBack**-nél az adatobjektum adatkötött tulajdonságának típusa.
- **object parameter**: esetleges paraméterek.
- **CultureInfo culture**: kultúra információ lokalizációhoz.

Adjunk a projekthez egy új osztályt, nevezzük el **KepKonverter** -nek és töltsük fel az alábbi kóddal:

```

public class KepKonverter: IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
        CultureInfo culture)
    {
        nem n = (nem)value;
        string img_src = string.Empty;
        switch (n)
        {
            case nem.Ferfi:
                img_src = "imgs/ferfi.png";
                break;

```

```
        case nem.No:
            img_src = "imgs/no.png";
            break;
        default: break;
    }
    return img_src;
}

public object ConvertBack(object value, Type targetType, object parameter,
    CultureInfo culture)
{ throw new NotImplementedException(); }
}
```

Két megjegyzésem van ehhez a kódrészlethez. Az egyik, hogy a **Convert** metódusok általában *switch/case* vezérlési szerkezetet szoktak használni. A másik pedig, hogy a **ConvertBack** metódust csak kétirányú adatkötés esetén kell megírni. A többi esetben a fentebbi kódrészletnél is használt **NotImplementedException** kivételdobást szokás meghagyni.

A konverterek példányosítása kicsit trükkösen történik. Elsőre azt gondolhatnánk, hogy az adatforráshoz hasonlóan, a **Loaded** eseményben be lehet állítani egy konvertert a **UserControl Converters** gyűjteményén keresztül, de a valóságban a **UserControl**-nak nincs ilyen tulajdonsága. Sőt, explicit módon egyetlenegy olyan tulajdonsága sincs, mely **IValueConverter**-eket várna. Emiatt nem C# kódból fogjuk példányosítani (bár nem lehetetlen vállalkozás), hanem XAML kódból. Statikus erőforrásként kell létrehozunk belőle egy példányt, amelyre majd hivatkozni lehet az adatkötésnél. De ahhoz, hogy példányosítani tudjuk, egy XML névtér deklarációra („XAML using”-ra) van szükségünk, amely az alkalmazás assembly főnévtérére mutat. Ezt általában „**this**” névvel szokás illetni, de bármilyen tetszőleges név választható helyette. A mi esetünkben ez az alábbi módon néz ki a **UserControl** nyitó tagén belül (a **hatterkonverter** miatt ez már itt van):

```
<UserControl x:Class="DataBinding_Demo.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    ...
    xmlns:this="clr-namespace:DataBinding_Converter_Demo"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
```

Ezek után már a **this**-en keresztül létre tudunk hozni egy **KepKonverter** példányt a **UserControl Resources** gyűjteményén belül. Adjuk neki a „**kepkonverter**” nevet!

```
mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
<UserControl.Resources>
    <this:KepKonverter x:Key="kepkonverter" />
</UserControl.Resources>
<Grid x:Name="LayoutRoot">
```

A konverterek használata egy rendkívül egyszerű feladat, ugyanis egy statikus erőforrásra kell hivatkozni egy **StaticResource markup extension** segítségével (ugyanúgy, mint a stílusoknál). Az adatkötés Binding-jának van egy **Converter** tulajdonsága, mely pont egy ilyen példányt vár:

```
<Image Source="{Binding Nem, Converter={StaticResource kepkonverter}}" />
```

A 6-9 ábrán látható, hogy nemcsak egyetlen adatforrást használunk, hanem többet is (lásd a projekt **MainPage.xaml.cs** fájl **Loaded** eseménykezelő függvénye). Ebben a demóalkalmazásban a névjegykártya felületét leíró kódrészlet is kétszer szerepel egymás alatt. Ez nem túl jó megoldás, a későbbiekben foglalkozunk majd azzal, hogy miként érdemes adathalmazok esetén a felhasználói felületet megszerkeszteni. Most viszont nézzük meg, hogy miként lehetséges két vezérlőt összekapcsolni XAML kódból!

Vezérlők közötti adatkötés

Az adatkötésnek kétségtelenül az egyik legjobb tulajdonsága az, hogy szintisztán deklaratív módon leírható. Ha ehhez még hozzávesszük a változáskövetési lehetőséget is, akkor egy olyan rendszert kapunk, amelyben egyetlen kódsor nélkül írhatunk le komponensek közötti függőségeket. Vagyis nem kell feliratkoznunk egy adott vezérlő valamely változást jelző eseményére ahhoz, hogy az őt lekezelő függvényben módosítsuk egy másik vezérlő valamely tulajdonságát, hanem mindezt meg tudjuk tenni XAML kódból. Ez a lényege a vezérlők közötti adatkötésnek, melyet angolul *Element To Element Binding*-nak szoktak nevezni.

A használata oly módon történik, hogy az adatkötni kívánt tulajdonságnál először megadjuk a forrásként szolgáló vezérlő megfelelő tulajdonságának a nevét, majd pedig az „ElementName=” kulcsszó után a vezérlő azonosítóját. Pl.: **ScaleX="{Binding Value, ElementName=MyHorizontalSlider}"**. Ebben a példában egy csúszka aktuális pozícióját kapcsoljuk össze az adott vezérlő X tengely (vagyis a vízszintes) menti nyújtási transzformációjával. Ezek után nézzük meg, miként lehet ezt egyszerűen megvalósítani grafikus felület segítségével Visual Studióban, illetőleg Expression Blendben.

Kiindulási alapként hozzunk létre egy új projektet a Visual Studióban **ElementsBinding_Demo** néven! A **LayoutRoot**-on belül definiáljunk két oszlopot és két sort az alábbi módon:

```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="0.2*" />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition Height="0.2*" />
  </Grid.RowDefinitions>
</Grid>
```

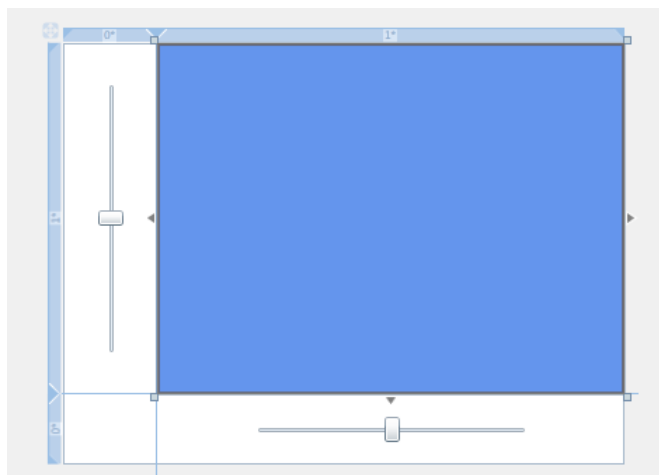
Ha ezzel megvagyunk, akkor helyezzünk el két csúszkát a **Grid**-en belül, egyet a **(0, 0)** cellába függőlegesen, egyet pedig az **(1, 1)** cellába vízszintesen! A kódjuk az alábbi módon áll össze:

```
<Grid x:Name="LayoutRoot" Background="White">
  ...
  <Slider x:Name="MyVerticalSlider" Orientation="Vertical" IsDirectionReversed="True"
    Maximum="1" Value="0.5" HorizontalAlignment="Center"
    VerticalAlignment="Center" Height="200" />
  <Slider x:Name="MyHorizontalSlider" Maximum="1" Value="0.5"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    Width="200" Grid.Column="1" Grid.Row="1" />
</Grid>
```

Végezetül pedig adjunk a felületünkhöz egy egyszerű téglalapot egy **ScaleTransform**-mal felvértézve a **(0, 1)** cellába, és töltsük ki valamilyen színnel, például kékkel! A transzformáció középpontját pedig állítsuk be a téglalap középpontjára!

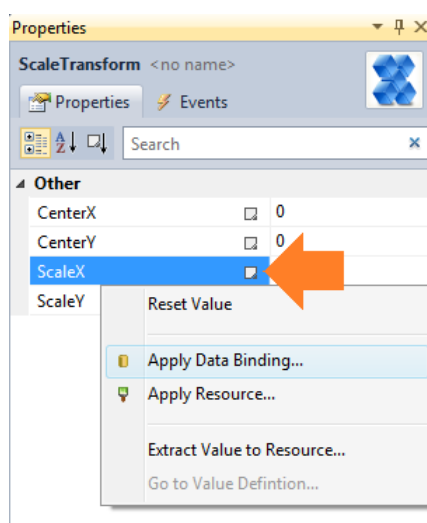
```
<Grid x:Name="LayoutRoot" Background="White">
  ...
  <Slider x:Name="MyHorizontalSlider" ... />
  <Rectangle Fill="CornflowerBlue" Grid.Column="1" RenderTransformOrigin="0.5,0.5" >
    <Rectangle.RenderTransform>
      <ScaleTransform />
    </Rectangle.RenderTransform>
  </Rectangle>
</Grid>
```

A tervezési időben látható felületnek a 6-10 ábrán látható módon kell kinéznie.



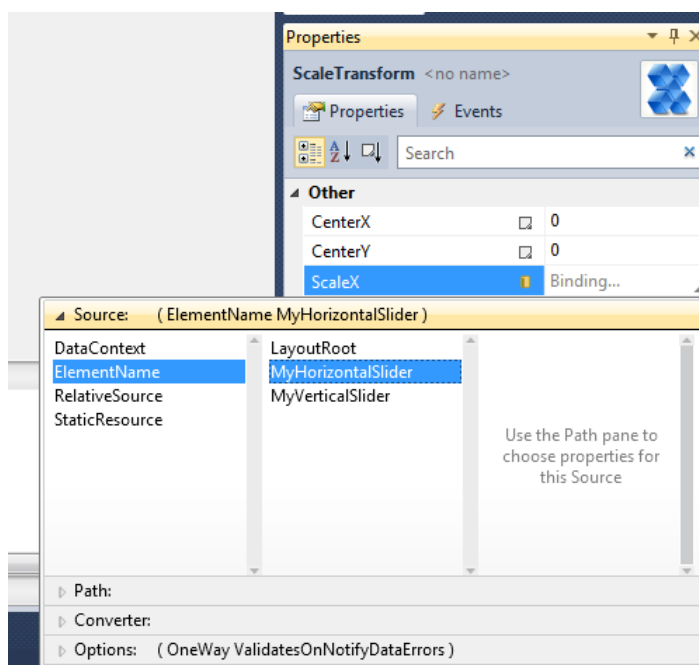
2-10 ábra: A vezérlők közötti adatkötéshez készített felület

Most kattintsunk bele a **ScaleTransform** kódjába, és nyomjuk meg az F4 gombot, hogy a Properties ablakban ez legyen a kiválasztott objektum! Itt a **ScaleX** neve mellett lévő kis négyzetre kattunk rá, mely feldobja az *Advanced Options* nevezetű helyi menüt! Ebben az *Apply Data Binding...* elnevezésű menüpontot válasszuk! Ezt a lépést a 6-11 ábra szemlélteti.



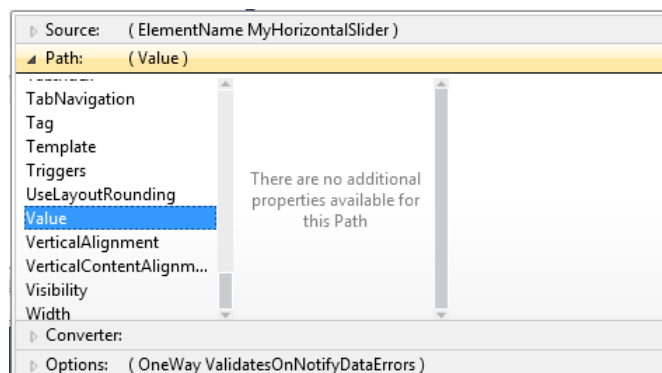
6-11 ábra: Advanced Options ➤ Apply Data Binding...

A felugró ablakban válasszuk ki bal oldalt az **ElementName** lehetőséget, majd a középső szektorból pedig a **MyHorizontalSlider** vezérlőt! A 6-12 ábrán tekinthető meg ez a lépés.



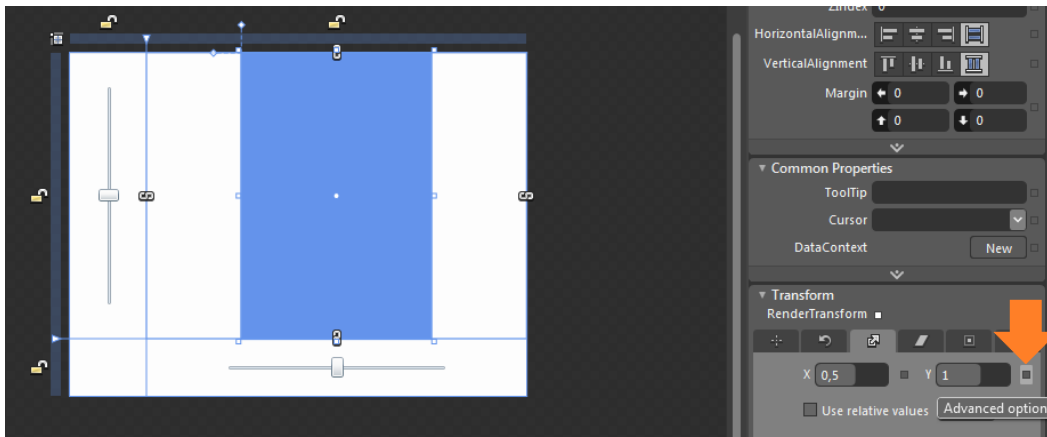
6-12 ábra: Baloldalt az ElementName-t, középen a MyHorizontalSlider-t válasszuk

Ezek után alul kattintsunk a *Path*: csíkra, és itt az első oszlopban a **Value** értéket keressük meg! A 6-13 ábrán látható ez a művelet.



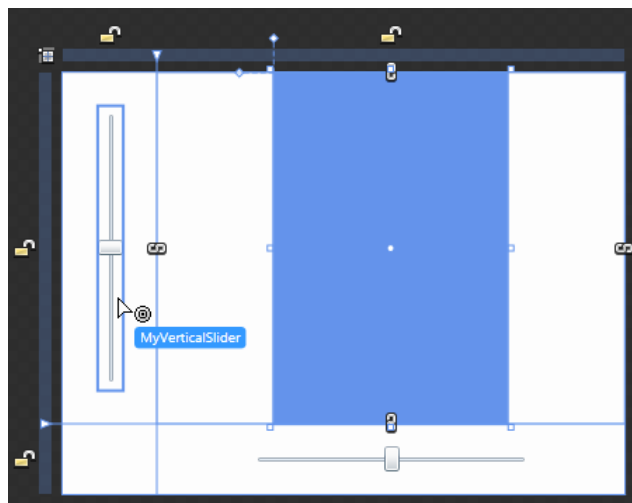
6-13 ábra: Path-nek a Value tulajdonságot válasszuk

A **ScaleX** adatkötésével meg is volnánk. Most nézzük meg a **ScaleY** adatkötését Expression Blend alatt! Kattintsunk a jobb egérgombbal a **MainPage.xaml**-re, majd a helyi menüből válasszuk ki az *Open in Expression Blend* menüpontot! Miután a Blend betöltődött, válasszuk ki az Objects and TimeLine ablakban a téglalapot, és a Properties ablakban keressük meg a **ScaleTransform** fület a Transforms csoporton belül! Itt kattintsunk az Advanced Options-t felugrasztó kis négyzetre! Ezt a lépést a 6-14 ábra szemlélteti.



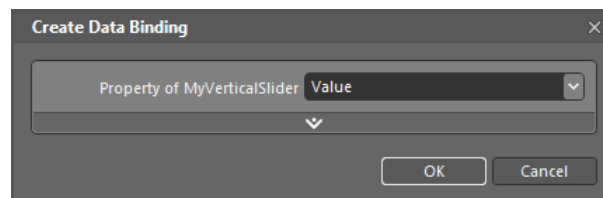
6-14 ábra: Expression Blend-ben az adatkötés

Itt válasszuk ki az *Element Property Binding* lehetőséget, majd vigyük az egerünket a **MyVerticalSlider** vezérlő fölé! (Ilyenkor az egér mellett megjelenik néhány kis koncentrikus kör. Ez a vezérlő választó jel a Blendben.) A 6-15 ábrán látható a vezérlő kijelölése.



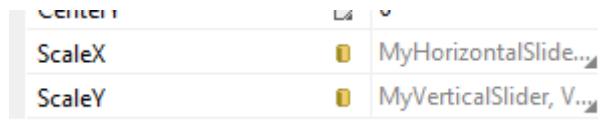
6-15 ábra: A MyVerticalSlider kijelölése

Az újonnan felugró ablakban alpból a **Value** tulajdonság lesz kiválasztva, ezért itt kattintsunk az OK gombra (6-16 ábra)!

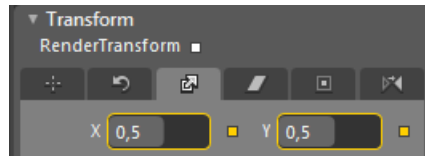


6-16 ábra: A Value tulajdonság kiválasztása

És ezzel el is készültünk. A Visual Studióval egyetemben az Expression Blend is jelzi, hogy a **ScaleX** és **ScaleY** tulajdonságok az értéküket adatkötésből nyerik, amint azt a 6-17 és 6-18 ábrák mutatják.



6-17 ábra: Visual Studio-s adatkötés figyelmeztetés



6-18 ábra: Blend-es adatkötés figyelmeztetés

Utolsó lépésként futtassuk az alkalmazást és próbáljuk ki!

A vezérlők közötti adatkötésnek van egy speciális esete, amikor egy **ContentTemplate**-en belül szeretnénk használni. Ilyenkor a **Binding RelativeSource** tulajdonságára van szükségünk, mely segítségével az éppen felül definiált vezérlő tulajdonságait érhetjük el a sablonon belül. (Ez annyival tud többet a **TemplateBinding**-nál, hogy míg az csak **OneWay** típusú, addig ez támogatja a **TwoWay**-t is.) Bővebb információt a **RelativeSource**-ról az alábbi cikkben találhattok: <http://www.wintellect.com/CS/blogs/jprosis/archive/2009/11/06/silverlight-3-s-new-relativesource-markup-extension.aspx>. Szintén a **RelativeSource** segítségével oldható meg az a feladat is, amikor a forrás és a cél ugyanaz a vezérlő.

Listás adatmegjelenítés

A listás adatmegjelenítés alapjai

Amint azt már a bevezetőben is elkezdtem boncolgatni, Silverlight esetén egyetlen objektum megjelenítése és egy adathalmaz vizualizációja nem kíván merőben másfajta gondolkodásmódot. Ha az adathalmazra úgy tekintünk, mint azonos típusú elemek egy listájára, akkor egyértelműen látszik, hogy bőven elegendő egy elem kinézetét megadni, és ezt kell csak sokszorozni valamilyen elrendezési logika szerint. Vagyis úgymond az összetett adatmegjelenítés visszavezethető az előző alfejezetekben bemutatott technikára.

Egy sablonon belül — konkrétan az **ItemTemplate**-en belül — kell megadnunk, hogy egy adott elem, rekord miként jelenjen meg a felhasználói felületen. Itt a tulajdonságokat, cella értékeket reprezentáló vezérlőket egyrészt adatkötjük, másrészt pedig egy konténervezérlő alá soroljuk be, ugyanis a sablonnak csak egyetlen gyerek eleme lehet. Pontosabban az **ItemTemplate**-en belülre először egy **DataTemplate**-t kell elhelyezni, és csak utána jöhet a konténervezérlő a gyerek elemeivel. Íme, egy egyszerű példa:

```
<ListBox>
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding Keresztnev}" />
        <TextBlock Text="{Binding Vezeteknev}" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Többféle listás adatmegjelenítésre képes vezérlő is található a Silverlightban, melyek között az alapvető különbség az elemeiknek az egymáshoz viszonyított helyzete. Ha a **ListBox** vezérlőt választjuk, akkor a

rekordok alából egymás alatt, függőlegesen felsorolva jelennek meg, viszont ha ezen az elrendezésen módosítani szeretnénk, akkor az **ItemsPanel** tulajdonságán belül egy **ItemsPanelTemplate**-n keresztül tudjuk ezt megtenni. Íme, egy egyszerű példa:

```
<ListBox>
  <ListBox.ItemsPanel>
    <ItemsPanelTemplate>
      <StackPanel Orientation="Horizontal" />
    </ItemsPanelTemplate>
  </ListBox.ItemsPanel>
</ListBox>
```

Egy másikfajta listás megjelenítő a **PathListBox**, mely alából nem része a Silverlight vezérlőparkjának, sőt nem is a Silverlight Toolkit-ben kapott helyett, hanem az Expression Blend kiegészítései között, konkrétan a **Microsoft.Expression.Controls.dll**-ben. Ez oly módon működik, hogy a **ListBox** elemeit egy **Path** objektumra fűzi fel, így adva a designernek szabad kezet a kinézet megkonstruálásában. Ezzel a vezérlővel ebben a fejezetben nem foglalkozunk, de érdemes lehet utánanézni az alábbi linken a vezérlő által nyújtott lehetőségeknek: <http://www.microsoft.com/design/toolbox/tutorials/pathlistbox/>.

Az egyszerű adatkötéssel ellentétben itt nem a **DataContext** tulajdonságnak kell értékül adni az adathalmazt, hanem az úgynevezett **ItemsSource**-nak.

Mellesleg itt a WinForms-os terminológiától eltérően a „kiválasztott elem megváltozott” eseményt nem **SelectedItemChanged**-nek hívják, hanem **SelectionChanged**-nek.

ObservableCollection használata mint adatforrás

A változásokövetés szuper dolog, főleg **OneWay** vagy **TwoWay** típusú adatkötéssel párosítva. Egyetlen objektum esetén ehhez az **INotifyPropertyChanged** interfészt kellett megvalósítanunk. De mi a helyzet a gyűjteményekkel? Mi van akkor, ha arról szeretnénk értesítést kapni, hogy egy új elem került be a kollekcióba vagy éppenséggel el lett távolítva? Nos, ebben az esetben az **INotifyCollectionChanged** interfészre lesz szükségünk, mely a **System.Collections.Specialized** névtérben található.

```
public interface INotifyCollectionChanged
{
    event NotifyCollectionChangedEventHandler CollectionChanged;
}
```

Látható, hogy az interfész az **INotifyPropertyChanged**-hez hasonlóan egyetlen eseményt tartalmaz. Ennek egy speciális **EventArgs** objektuma van, mely a **NotifyCollectionChangedEventArgs** névre hallgat. Ennek az objektumnak egyetlen fontos tulajdonsága van, az **Action**. Ez egy olyan felsorolt típus, mely segítségével meg tudjuk mondani a rendszernek, hogy milyen típusú változás váltotta ki ezt az eseményt. (Az enum az alábbi értékek valamelyikét veheti fel: **Add**, **Remove**, **Replace**, **Reset**). Íme, az interfész általános megvalósítása:

```
public class MyCollection: INotifyCollectionChanged, IEnumerable, IEnumerator
{
    ...
    public event NotifyCollectionChangedEventHandler CollectionChanged;
    protected virtual void OnCollectionChanged(NotifyCollectionChangedEventArgs e)
    {
        if (CollectionChanged != null)
            CollectionChanged(this, e);
    }
}
```



```

public void Add(Object o)
{
    ...
    OnCollectionChanged(
        new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Add));
}
public void Remove(Object o)
{
    ...
    OnCollectionChanged(
        new NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Remove));
}

...

#region IEnumerable Members ... #endregion

#region IEnumerator Members ... #endregion
}

```

Ez a kódrészlet eléggé hosszúra sikeredett, még így is, hogy nagyon sok részletet (például a gyűjtemény reprezentációt, az **IEnum*** interfészek megvalósítását) elhagytam belőle. Az osztály teljes kódja elérheti akár a 200 sort is, és nem a reprezentáció bonyolultsága miatt, hanem azért, mert minden **IEnumerable** és **IEnumerator** műveletet nekünk kell megírunk. Ha ezt minden egyes gyűjtemény esetén be kellene gépelnünk, akkor elég hamar eljutnánk oda, hogy íránk belőle egy általánosabb, generikus változatot. A jó hír az, hogy erre nincs szükség, ugyanis a Silverlight fejlesztői elkészítették ezt helyettünk. Ezt a generikus osztályt a nagyon találó **ObservableCollection** névre keresztelték, és a **System.Collections.ObjectModel** névtérben helyezték el.

A dolog szépsége, hogy nincs szükségünk explicit adatkötésre a felhasználói felület frissítéséhez. Elegendő, ha az **ItemsSource** tulajdonságnak a kódból értékül adjuk az **ObservableCollection**-t. Vagyis nem kell XAML kódból adatkötni a gyűjteményt ahhoz, hogy legyen változáskövetésünk. (Lásd következő alfejezetbeli példa).

Listás adatmegjelenítés Visual Studióban

Ebben a részben kétféle technikát fogok bemutatni Visual Studio 2010-ben, melyek között az alapvető különbség a példaadatok tárolásának és az adatforrás megadásának a módja. Először nézzük az egyszerűbbet!

Hozzunk létre egy új projektet, és adjuk neki a **ListBased_DataVisualization_Demo** nevet! Adjuk hozzá a Silverlightos alkalmazáshoz azt a **Dolgozo.cs** fájlt, melyet a *Konverterek használata* című alfejezetben használtunk (a névteret ne felejtsük el átírni)! Ezek után adjunk egy újabb osztályt a projekthez **Dolgozok.cs** néven, és az osztályt származtassuk az **ObservableCollection<Dolgozo>** típusból!

```

public class Dolgozok: ObservableCollection<Dolgozo>
{ }

```

Erre az osztályra azért van szükségünk, hogy XAML kódból statikus erőforrásként létre tudjunk hozni mintaadatokat az egyszerűbb és kényelmesebb felhasználói felület tervezéshez. A **MainPage.xaml UserControl** részében regisztráljuk be *this* néven az alkalmazás assemblyt!

```

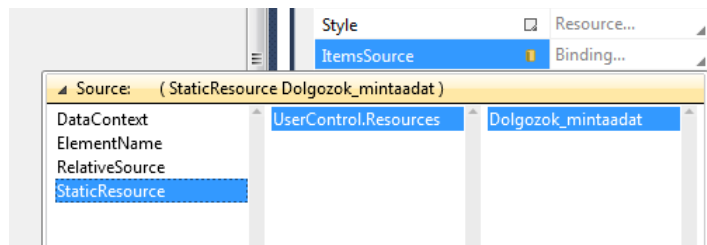
<UserControl x:Class="ListBased_DataVisualization_Demo.MainPage"
    ...
    xmlns:this="clr-namespace:ListBased_DataVisualization_Demo"
    mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">

```

Ezek után pedig statikus erőforrásként hozzunk létre néhány **Dolgozo** objektum példányt!

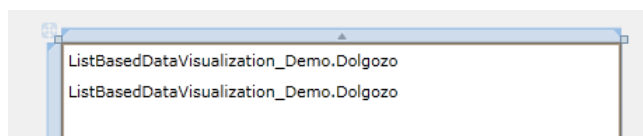
```
mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
<UserControl.Resources>
  <this:Dolgozok x:Key="Dolgozok_mintaadat">
    <this:Dolgozo Vezeteknev="Dotnet" Keresztnev="Elek"/>
    <this:Dolgozo Vezeteknev="Szájhar" Keresztnev="Mónika"/>
  </this:Dolgozok>
</UserControl.Resources>
```

Adjunk a **LayoutRoot**-hoz egy **ListBox** vezérlőt és a Properties ablakban az **ItemsSource** tulajdonságánál válasszuk ki az adatkötést (Advanced Options ➤ Apply Data Binding)! Itt bal oldalt válasszuk ki a **StaticResource** lehetőséget, középen a **UserControl.Resources**-t, jobb oldalt pedig a **Dolgozok_mintaadat**-ot! Ezt a lépést a 6-19 ábra szemlélteti.



6-19 ábra: *ItemsSource* adatkötése egy statikus erőforráshoz

Ha most megnézzük a Silverlight Designer felületen a **ListBox**-ot, akkor a 6-20 ábrához hasonlót kell látnunk.

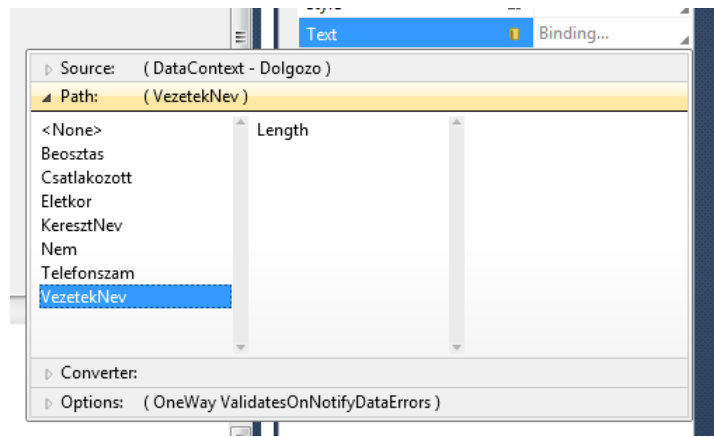


6-20 ábra: A **ListBox** az **ItemsSource** tulajdonság beállítás után

Mivel még nem állítottuk be a **ListBox** **ItemTemplate** tulajdonságát, ezért az egyes elemek a **ToString()** függvényük által visszaadott formájukban jelennek meg. Sajnos Visual Studio 2010-ben nincs arra lehetőség, hogy a designer felületen keresztül pakoljuk össze az **ItemTemplate**-en belüli sablont. Viszont kapunk IntelliSense támogatást a XAML kódban az adatkötéskor a tulajdonságok kiválasztásához, illetve a Properties ablakban is láthatóak az adatforrás választható tulajdonságai. Ezek után írjuk be az alábbi pár sort a **ListBox** vezérlőn belülre:

```
<ListBox ItemsSource="{Binding Source={StaticResource Dolgozok_mintaadat}}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock />
        <TextBlock />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Válasszuk ki az első **TextBlock**-ot, majd a Properties ablakban adatkössük a **Text** tulajdonságát az adatforrás **Vezeteknev** tulajdonságához! Ezt a lépést a 6-21 ábra szemlélteti.



6-21 ábra: VezetekNev tulajdonság adatkötése az első TextBlock Textjéhez

A **Keresztnev**-et hasonlóképpen kapcsoljuk össze a második **TextBlock**-kal. A 6-22 ábra azt mutatja, amit az adatkötés után a tervezőfelületen láthatunk.



6-22 ábra: Design-Time támogatás a ListBox megjelenítéséhez

A 6-22 ábrán látható, hogy a Keresztnev közvetlenül ott lohol a Vezetéknév nyomában. Az elválasztáshoz szükséges helyet valamelyik **TextBlock Margin** tulajdonságának beállításával tudjuk biztosítani.

Természetesen, ha kódból valamilyen adatforrást hozzárendelünk az **ItemsSource** tulajdonsághoz, akkor az fog megjelenni futásidőben.

```
public MainPage()
{
    InitializeComponent();
    lb_dolgozok.ItemsSource = new Dolgozok
    {
        new Dolgozo { Vezeteknev="Teszt", Keresztnev="Elek"},
        new Dolgozo { Vezeteknev="Teszt", Keresztnev="Elekné"}
    };
}
```

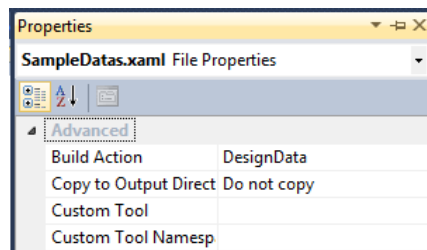
Ez volt az első módszer, amelynek van egy apró kis problémája: a felületet leíró kód keveredik a mintaadattal, és ez így nem túl szép. A másik technika (mely az előző továbbfejlesztése) ezt a szépséghibát hivatott kijavítani.

Adjunk a projekthez egy új mappát **DesignData**s néven! Ezen belül hozzunk létre egy új szövegfájl, és nevezzük el **SampleData.sxaml**-nek! Az alábbi kódrészletet helyezzük el benne:

```
<this:Dolgozok xmlns:this="clr-namespace:ListBased_DataVisualization_Demo">
  <this:Dolgozo Vezeteknev="Dotnet" Keresztnev="Elek"/>
  <this:Dolgozo Vezeteknev="Szajhar" Keresztnev="Mónika"/>
</this:Dolgozok>
```

Ez ugyanaz a kódrészlet, mint ami a **UserControl Resources** gyűjteményén belül található, kivéve, hogy ennek nincs kulcsa, ellenben van egy névtér regisztrációja.

Állítsuk be a fájl **Build Action**-jét **DesignData**-ra, illetve győződjünk meg róla, hogy a **Copy to Output Directory** tulajdonság **Do not copy**-ra van állítva! Végezetül a **Custom Tool** tulajdonság értékét töröljük ki! A 6-23 ábra ezt szemlélteti.



6-23 ábra: A *SampleData.xml* tulajdonságai

A következő lépés, hogy a **MainPage.xaml**-ben lévő **UserControl Resources** gyűjteményét töröljük ki. Ezek után pedig keressük meg a **ListBox** vezérlőt és a nyitó tagját az alábbira módosítjuk:

```
<ListBox d:DataContext="{d:DesignData Source=../DesignData/SampleData.xml}"
ItemsSource="{Binding}">
```

A **d** névtérben lévő összes *attached property*, *markup extension*, stb. design-time információval szolgál az Expression Blend és a Visual Studio XAML Designerének, vagyis ezek futásidőben nem érhetőek el. (Az előző változatnál, ha nem definiáljuk felül kódból az **ItemsSource**-t, akkor a mintaadatok fognak megjelenni futásidőben is, míg itt csak és kizárólag tervezési időben láthatóak a mintaadatok!)

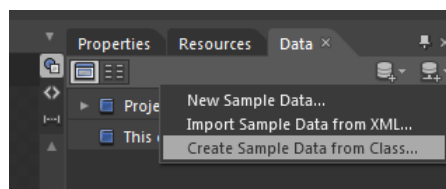
Az **ItemsSource** tulajdonságnál lévő kód elsőre kicsit furcsa lehet, de ez mindösszesen csak annyit jelent, hogy a teljes **DataContext**-beli objektum az adatforrás. A mintaprogramunk ezzel elkészült.

Listás adatmegjelenítés Expression Blendben

Az Expression Blend valamivel jobban segíti a munkánkat, ugyanis van néhány olyan rendkívül hasznos szolgáltatása, melyek nem találhatók meg a Visual Studio 2010-ben, például mintaadat generálás és kinézet szerkesztés.

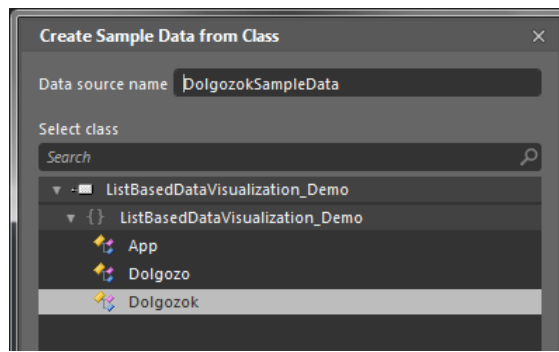
A mintaadatok generálásához ugyanazt a technikát használja a Blend, mint amit az előző mintaprogramnál láthattunk. Itt azonban a Blend állítja elő a tervezési időben használt adatok – a **Dolgozo** osztály példányait. (Megjegyezném, hogy utólag mi is módosíthatjuk az általa generált mintaadatokat, melyeket a **SampleData/DolgozokSampleData.xml** fájlban sorol fel.)

A **LayoutRoot**-on lévő **ListBox**-ot kommentezzük vagy töröljük ki, majd nyissuk meg az alkalmazást Blendben! (**MainPage.xaml**-en jobb klikk ➤ Open in Expression Blend). Ezután keressük meg a **Data** ablakot és itt kattintsunk jobbról a második ikonra, vagyis a *Create sample data*-ra! A lenyíló menüben válasszuk a *Create Sample Data from Class* lehetőséget! (6-24 ábra)



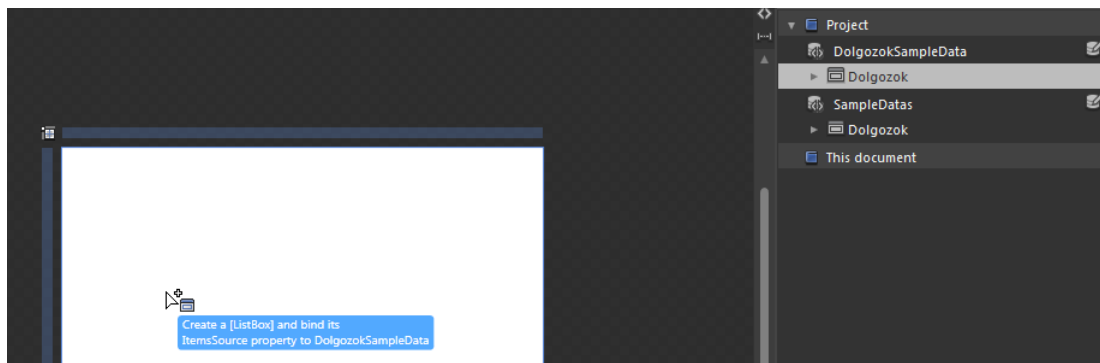
6-24 ábra: Mintaadat generálás kiválasztása

Az itt felugró ablakban válasszuk ki a **Dolgozok** osztályt! (6-25 ábra)



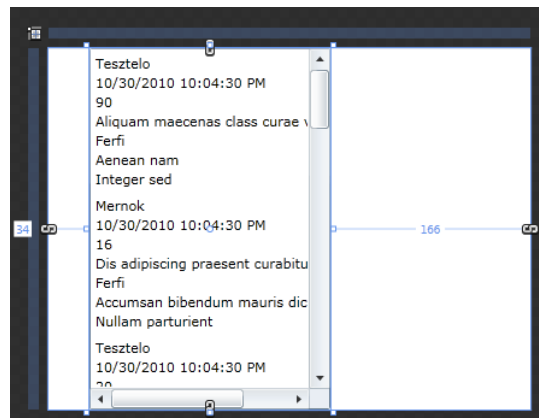
6-25 ábra: Dolgozok osztály kiválasztása

A Data ablakban meg fog jelenni egy **DolgozokSampleData** objektum. Ennek a Dolgozok tulajdonságát fogjuk meg egérrel, és húzzuk a felületre! (6-26 ábra)



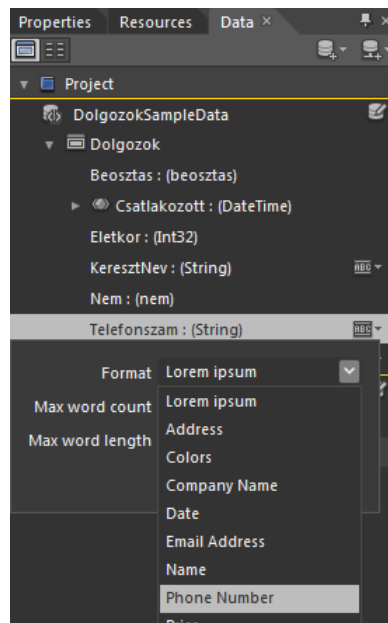
6-26 ábra: A Dolgozok ráhúzása a felületre

Miután ráhúztuk, egy **ListBox** fog megjelenni, telis-tele adatokkal. (6-27 ábra)



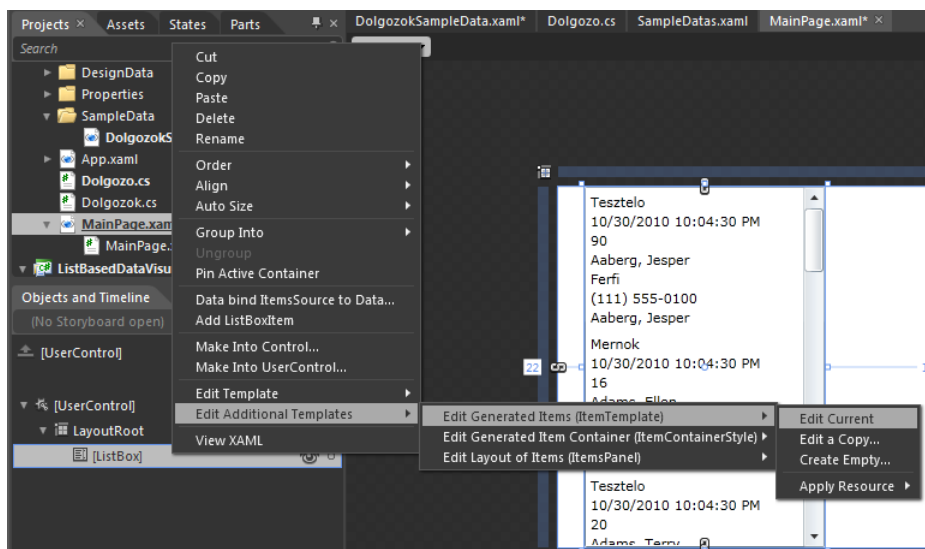
6-27 ábra: ListBox használata a mintaadatok reprezentálásához

Ez már majdnem jó, de mi azt szeretnénk, ha a kereszt- és vezetéknév nem mondatok lennének, és a telefonszám is telefonszám lenne. Ezeket mind-mind be tudjuk állítani a **DolgozokDataSource**-n, ha kinyitjuk a **Dolgozok** részt, és az adott tulajdonságok mellett lévő ABC ikonra kattintunk. Itt a **Format** tulajdonságon keresztül tudjuk kiválasztani a nekünk megfelelő formátumot. (6-28 ábra)



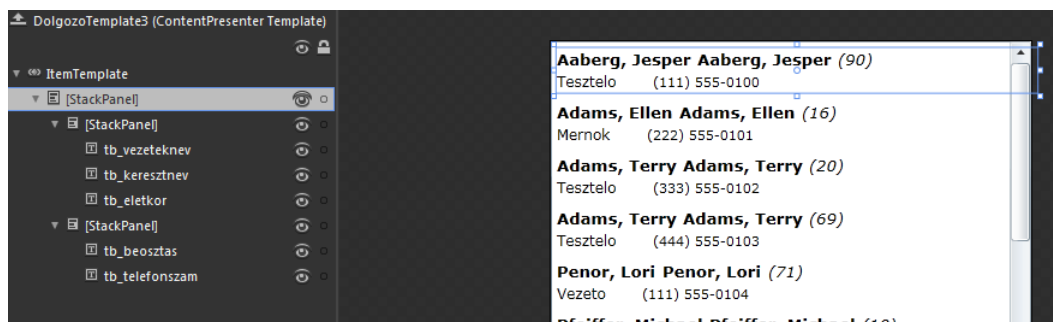
6-28 ábra: Megfelelő formátum kiválasztása

Ezek után javítsunk egy kicsit az egyes dolgozók kinézetén is, vagyis alakítsuk át a Blend által generált **ItemTemplate**-et! Az Objects and Timeline ablakban kattintsunk jobb egérgombbal a ListBoxra, majd válasszuk ki az *Edit Additional Templates* ➤ *Edit Generated Items (ItemTemplate)* ➤ *Edit Current* menüpontot! (6-29 ábra)



6-29 ábra: Az ItemTemplate módosítása

Itt alakítsuk át tetszés szerint a vezérlők kinézetét és elrendezését a grafikus felületen keresztül a Properties ablak segítségével! (6-30 ábra)

6-30 ábra: *ItemTemplate* módosítás után

Látható, hogy ez sem tökéletes, mivel a **Name** formátum keresztnév + vezetéknév párból áll, de ezzel talán együtt tudunk élni. Az Expression Blend mindenesetre nagyságrendekkel megkönnyíti a listaelemek kinézetének szerkesztését és a mintaadatok használatát, generálását.

Az adathalmazok igazi nemezése a *Master/Details* megjelenítés, melyhez a Silverlightban nagy segítséget nyújt számunkra a **CollectionViewSource**. Az alábbi oldalon találhattok egy példát a használatára: [http://msdn.microsoft.com/en-us/library/cc645060\(v=VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc645060(v=VS.95).aspx).

Összefoglalás

Ebben a fejezetben áttekintettük az adatkötés alapjait, melynek segítségével deklaratív módon lehet összekapcsolni az adatréteget a megjelenítési réteggel. Megnéztük, milyen előnyökkel jár az automatikus változásokövetés és a kétirányú adatkommunikáció. Továbbá utánajártunk annak is, miként lehet az adatkötést testreszabni és vezérlők között alkalmazni. Végezetül pedig bepillantást nyerhettünk a konverterek megvalósításába és az adathalmazok megjelenítésébe.

7. Saját vezérlők készítése Silverlightban

Magam részéről mindig is úgy gondoltam, hogy saját vezérlőket írni művészet. Egy komplex vezérlő kivitelezése, a legtöbb platform esetében, rendkívül átfogó ismereteket igényel, bizonyítja készítőjéről, hogy a kérdéses platform „nyelvét” és technikáit jól ismeri, és helyesen alkalmazza. Nincs ez másképp a Silverlight esetében sem. A semmiből bonyolult, komoly vizualizációval rendelkező vezérlőt, vezérlőket csak azok tudnak készíteni, akik igazán értik, hogy is működik a Silverlight platform. Ezért is olyan fontos ennek a témának az alapos megismerése, elsajátítása.

Kétségkívül kijelenthetjük, a vezérlők testreszabhatósága az egyik legizgalmasabb terület Silverlightban. Az új vezérlőmodellnek köszönhetően egészen különleges lehetőségek kínálkoznak a korábbi vezérlők újragondolásakor. Így születhet **ProgressBar**-ból az autók kilométerórájára emlékeztető vezérlő, vagy egyszerű **ToggleButton**-ból repülő irányítópultján található kapcsolóhoz hasonló kétállású gomb.

Ennek a rugalmasságnak köszönhetően nem lesz szükségünk új vezérlő írására, amennyiben csak a vizualitást akarjuk átalakítani kívánalmaink szerint. Azonban ha a külsőn kívül a funkciókhoz is szeretnénk hozzájárulni, bizony saját vezérlők írásába kell fognunk. Az esetek többségében meglévő vezérlők funkcióinak kiterjesztésekor elegendő csupán a kiszemelt vezérlőből származtatnunk, és a kívánt funkcionalitást hozzáírunk a leszármaztatott vezérlőhöz. Akárcsak a korábbi prezentációs technológiák, a Silverlight is hasonló koncepcióval rendelkezik ilyen téren.

Igazán izgalmas attól a ponttól lesz a dolog, amikor némi tanakodás után arra jutunk, hogy a meglévő vezérlők nem elégítik ki az igényeinket, vagy testreszabásuk és a leszármaztatás együttes kombinációi túlságosan bonyolult feladatot jelentenének, így úgy döntünk, hogy nulláról kezdünk saját vezérlő fejlesztésébe.

A Silverlight két lehetőséget biztosít számunkra: **UserControl**ok vagy **Custom Control**ok készítését.

Ebben a fejezetben megismerkedünk ezekkel a lehetőségekkel, továbbá megtanuljuk, hogy miként támogassuk azoknak a fejlesztőknek a munkáját, akik a későbbiekben szeretnék testreszabni az általunk készített új vezérlőket.

UserControlok fejlesztése Silverlightban

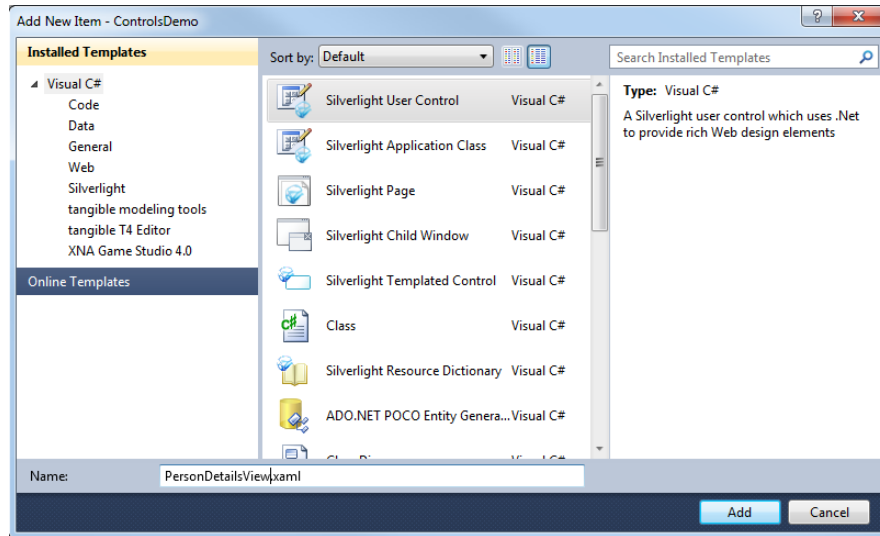
A UserControlok szerepe

Akárcsak a legtöbb prezentációs technológia esetén, így a Silverlightban is a **UserControl**ok nem mások, mint egyszerű vezérlők egyetlen komplex vezérlőbe csoportosítása valamilyen logika mentén, szem előtt tartva az újrafelhasználhatóság követelményeit. Csakhogy Silverlightban ez gyakorlatilag mindenre igaz. Minden egyes oldal a Silverlightban egy **UserControl** objektum. A **MainPage** osztály is egy **UserControl**. Ha netán Silverlight Navigation Applicationt hoztunk létre, akkor minden egyes oldal (pl. a **MainPage** osztály is) a **Page** ősosztályból származik, ami történetesen közvetlenül a **UserControl** leszármazottja. Jól látható tehát, hogy Silverlightban a **UserControl**ok kitüntetett szereppel bírnak, mindenhol körülvesznek minket. Általában más platformokon a fejlesztő azért készít **UserControl**t, hogy az újrafelhasználhatóságot biztosítsa. A Silverlightban az elsődleges szempont inkább a funkcionális egységbezárás (*encapsulation*). A legtöbb Silverlight fejlesztő nem szereti egyetlen oldalra ömleszteni vezérlők tucatjait és csupán panelek segítségével elszeparálni őket. A panelek bár csökkenthetik a kontrolok sokasága miatt okozott káoszt a vizuális fában, de a logikai szempontokat figyelembe véve a programozhatóság, a kezelhetőség továbbra is csorbul. Ennek megfelelően inkább a logikai elkülönítést részesítik előnyben. Úgy fogják fel a felhasználói felületet, mint önálló funkciókkal rendelkező nézetek összességét. Például egy ügyfélnyilvántartó rendszerben egy adott ügyfél adatai egyetlen nézetben

jelenhetnek meg, amit egy **UserControl** objektummal lehet reprezentálni. Még akkor is érdemes ezt az utat választani, ha a vezérlő újrafelhasználhatósága egyáltalán nem szempont.

UserControlok létrehozása

Új User Control objektumot a Visual Studio 2010-ben úgy adhatunk a projekthez, hogy a Solution Explorerben a jobb egérgombbal a projektre kattintunk, majd az Add New Item funkciót választjuk. A felugró ablakból a Silverlight User Control sablont kell kiválasztanunk (7-1 ábra).



7-1 ábra: UserControl hozzáadása a projekthez a Visual Studio 2010-ből

A vezérlő létrehozását követően jelen példában egy **PersonDetailsView.xaml** és egy **PersonDetailsView.xaml.cs** fájlt kaptunk. Az előbbi a **UserControl** felhasználói felületét írja le, az utóbbi a kapcsolódó mögöttes kódot tartalmazza. Ettől a ponttól ugyanúgy dolgozunk tovább, mint eddig, hiszen korábban is egy **UserControl** belsejében dolgoztunk, ami nem más volt, mint a **MainPage** osztály.

Az alábbi kódrészletben egy a **Person** osztály számára nézetként funkcionáló UserControlt láthatunk. Ezt a nézetet bárhol, bármilyen oldalon vagy bármilyen más nézet belsejében újra felhasználhatjuk. Sőt, ha a vezérlőt külön class libraryben helyezzük el, akkor más projektekben is felhasználhatóvá válik. A **PersonDetailsView** UserControl egy lehetséges implementációja látható az alábbi kódrészletben.

A **PersonDetailsView.xaml** tartalma:

```
<UserControl x:Class="ControlsDemo.PersonDetailsView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">
        <Grid.RowDefinitions>
            <RowDefinition Height="0.137*"/>
            <RowDefinition Height="0.15*"/>
            <RowDefinition Height="0.157*"/>
            <RowDefinition Height="0.403*"/>
            <RowDefinition Height="0.153*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.36*"/>
            <ColumnDefinition Width="0.64*"/>
        </Grid.ColumnDefinitions>
    </Grid>
</UserControl>
```

```

<TextBlock HorizontalAlignment="Left" TextWrapping="Wrap"
    VerticalAlignment="Center" Margin="8,0,0,0" FontWeight="Bold" Text="Name"/>
<TextBlock HorizontalAlignment="Left" TextWrapping="Wrap" Text="Address"
    Grid.Row="1" VerticalAlignment="Center" Margin="8,0,0,0" FontWeight="Bold"/>
<TextBlock HorizontalAlignment="Left" TextWrapping="Wrap" Text="On vacation?"
    Grid.Row="2" VerticalAlignment="Center" Margin="8,0,0,0" FontWeight="Bold"/>
<TextBlock HorizontalAlignment="Left" TextWrapping="Wrap" Text="Picture"
    Grid.Row="3" VerticalAlignment="Top" Margin="8,8,0,0" FontWeight="Bold"/>
<TextBox TextWrapping="Wrap" Text="{Binding Name, Mode=TwoWay}" Grid.Column="1"
    VerticalAlignment="Center" Margin="8,0,20,0"/>
<TextBox TextWrapping="Wrap" Text="{Binding Address, Mode=TwoWay}"
    Grid.Column="1"
    Grid.Row="1" VerticalAlignment="Center" Margin="8,0,20,0"/>
<CheckBox Content="" Grid.Column="1" Grid.Row="2" VerticalAlignment="Center"
    Margin="8,0,20,0" IsChecked="{Binding IsOnVacation, Mode=TwoWay}"/>
<Button Content="OK" Click="btn_SaveDataOnClick" Margin="10,7,0,17"
    Grid.Column="1" HorizontalAlignment="Left" Grid.Row="4" Width="75"/>
<Image Source="{Binding Picture}" HorizontalAlignment="Left"
    VerticalAlignment="Top" Width="100" Height="100" Grid.Column="1"
    Margin="8,6,0,0" Grid.Row="3"/>
</Grid>
</UserControl>

```

A **PersonDetailsView.xaml.cs** tartalma:

```

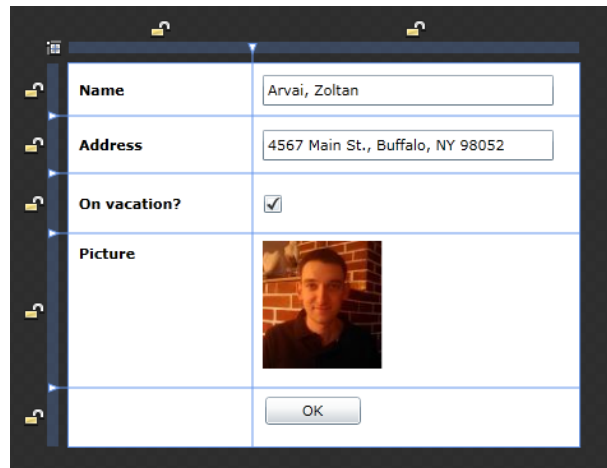
using System;
using System.Windows;
using System.Windows.Controls;

namespace ControlsDemo
{
    public partial class PersonDetailsView : UserControl
    {
        public PersonDetailsView()
        {
            InitializeComponent();
        }

        private void btn_SaveDataOnClick(object sender, RoutedEventArgs e)
        {
            //TODO
            //Save the data now
            //....
        }
    }
}

```

A kód által megjelenített felhasználói felület a 7-2-es ábrán látható. Ez egy egyszerű nézet, amely a **Person** osztály egyes tulajdonságait jeleníti meg szerkeszthető formában, majd az OK gombra kattintva a szükséges adatokat, változásokat elmenti.



7-2 ábra: A UserControl Designer nézetben

A **UserControl** belsejében jól látható hogy Bindingokat alkalmazunk a **Person** osztály egyes tulajdonságaira. Ez azt jelenti, hogy a **UserControl** felhasználáskor **DataContext**ben át kell adnunk egy **Person** példányt ahhoz, hogy helyesen működni tudjon. Ezt a **Person** példányt a **UserControl** tovább örökíti a benne található vizuális fában, a **DataContext** tulajdonságon keresztül.

User Controlok felhasználása

Az általunk készített **UserControl** felhasználásához hivatkoznunk kell rá. Ez a vezérlő nyilván egy másik névtérben található, mint az alapvető Silverlight vezérlők. Ezért definiálni kell egy prefixet, ami meghatározza, hogy az adott vezérlő melyik dll-ben és melyik névtérben található. Jelen esetben a **UserControl** ugyanabban a dll-ben van, mint a vezérlőt felhasználó projekt, így csak a névtérrel kell meghatározni az **xmlns:local="clr-namespace:ControlsDemo"** attribútum segítségével. Amennyiben a vezérlő egy másik assemblyben szerepel, akkor azt is jeleznünk kell a következőképpen:

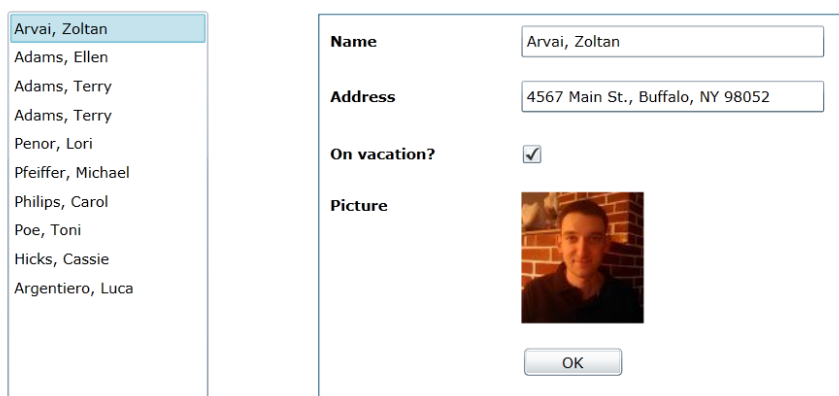
xmlns:local="clr-namespace:ControlsDemo;assembly=myassembly".

Az alábbi kódrészlet mutatja a **PersonDetailsView** vezérlő felhasználását.

```
<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:ControlsDemo" x:Class="ControlsDemo.MainPage"
  mc:Ignorable="d"
  d:DesignHeight="435" d:DesignWidth="624">
  <Grid x:Name="LayoutRoot" Background="White"
    DataContext="{Binding Source={StaticResource PersonDataSource}}">
    <ListBox x:Name="listBox" DisplayMemberPath="Name" ItemsSource="{Binding People}"
      Margin="15,53,0,83" HorizontalAlignment="Left" Width="153"/>
    <Border Margin="252,55,82,87" BorderThickness="1" BorderBrush="#FF1E6183" >
      <local:PersonDetailsView DataContext="{Binding SelectedItem,
        ElementName=listBox}" />
    </Border>
  </Grid>
</UserControl>
```

A **PersonDetailsView** vezérlőnk **DataContext** tulajdonságát adatkötöttük a **listBox** vezérlő **SelectedItem** tulajdonságához, így mindig a **ListBox**-ban kiválasztott **Person** elem lesz a nézet aktuális **DataContext**-je. Ezzel egyféle master-detail kapcsolatot készítettünk.

Az eredmény a 7-3 ábrán látható.



7-3 ábra: A kiválasztott Person objektum részleteinek megjelenítése a PersonDetailsView UserControlban

UserControl: Nézet vagy hagyományos vezérlő?

A fenti példa jól demonstrálja, hogy a UserControlok tökéletesen alkalmasak nézetek létrehozására. A fenti példakódot használhatjuk egy felugró ablakban, egy másik oldalon, bármilyen becsúszó panelen, akármilyen master-detail helyzetben, és így tovább. Ezen megközelítés szerint a UserControlokra leginkább úgy gondolhatunk, mint az adatainkat reprezentáló nagyobb nézetekre, felhasználói felület töredékekre.

Azonban a UserControlokat gyakran használjuk más helyzetekben is. Például, a fenti esetben használt **ListBox ItemTemplate**-jeként is definiálhattunk volna egy **UserControl**-t. Persze, most joggal tehetjük fel a kérdést, hogy miért ne lenne elegendő egy önmagában tisztán definiált, erőforrásként elhelyezett **DataTemplate**? A válasz egyszerű: Mi van akkor, ha az adott elem nemcsak adatot jelenít meg, hanem valamilyen funkciót is végrehajt? Például, a sablonban szerepel egy Remove gomb, amire rákattintva az adott elemet törli az adatbázisból. Jóval elegánsabb és egyszerűbb ezt a feladatot a UserControlhoz tartozó mögöttes kódból megoldani, ahol a Remove gomb eseménykezelőjében — ismerve az adott törölendő elemet — könnyedén futtathatunk bármilyen kódot.

Igen sokat emlegetjük a UserControlokkal kapcsolatban a „nézet” fogalmat. Tényleg csak erre lennének jól használhatók? Természetesen készíthetünk velük klasszikus vezérlőket is, így például a következő szekcióban szereplő vezérlőt is megírhatnánk UserControlként, de mielőtt belefognánk, alaposan mérlegelnünk kell az előnyöket és a hátrányokat.

- A **UserControl** sablonja (**ControlTemplate**) nem cserélhető, azaz a kinézet később nem módosítható, csak a **UserControl** teljes áttervezésével.
- Egyetlen projekten belül az adott **UserControl** mindenhol ugyanúgy néz ki, a kinézet csak egyszerre változtatható, külön-külön nem.
- **UserControl**-t írni kicsit könnyebb, mint Custom Controlt

Ha klasszikus vezérlők írásába szeretnénk fogni, kézenfekvőbb inkább Custom Control-t használni.

Custom Controlok fejlesztése Silverlightban

A saját vezérlők másik válfajáról, a Custom Controlokról nem volt még szó. Képzeljük el a következő igényeket!

Egy **NumericUpDown** vezérlőt szeretnénk készíteni. Ez a vezérlő, szerkeszthető módon mindig megjeleníti az aktuális, egész szám értéket, aminek adatköthetőnek kell lennie. Ezen felül rendelkezik két gombbal. Az egyik gomb az érték pozitív irányba történő növelésére, a másik a negatív irányba való csökkentésére szolgál. Amennyiben bármelyik gombra ráklikkel a felhasználó, a megfelelő, pozitív vagy negatív irányba változik az érték, és ez rögtön látszik a megjelenített értéken is. A vezérlő vizuálisan megkülönbözteti a pozitív és negatív értékeket. Alapértelmezetten a negatív értékek esetén a megjelenítő **Foreground**

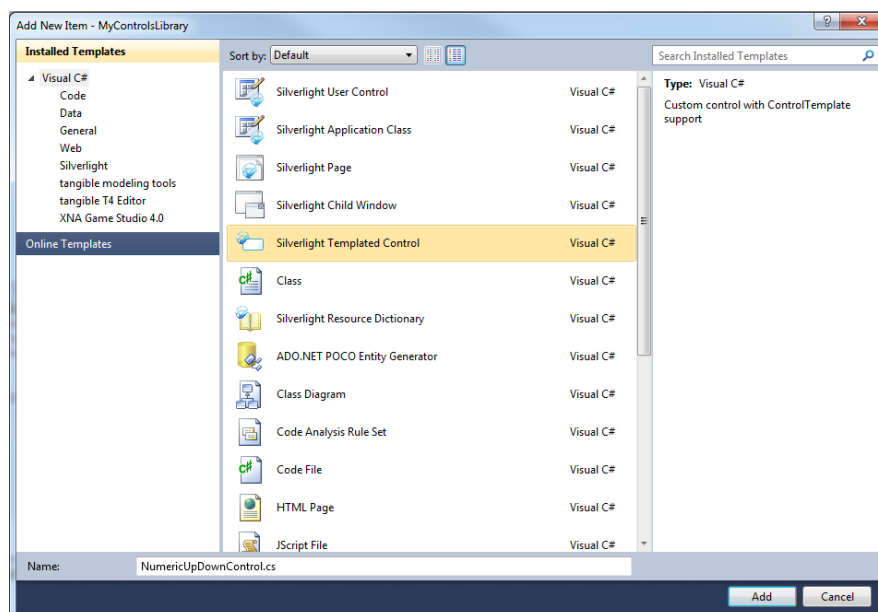
tulajdonsága kék, pozitív értékek esetén piros. *Elvárás, hogy a kinézetet később bármikor lehessen cserélni.*

Megjegyzés: Ezeket az igényeket az utolsó mondat kivételével egy **UserControl** segítségével pompásan meg tudtuk volna oldani. De az utolsó igény, a megjelenítés testreszabhatósága, a negatív és pozitív vizuális állapotok, az elrendezés és a vezérlők cseréje UserControlként már szinte megoldhatatlan feladat elé állított volna minket.

Ezeket a követelményeket kell kielégítenünk, és közben számos problémát kell leküzdenünk. A következő szekciókban ezen a példán fogunk végighaladni, és ennek a vezérlőnek a segítségével fogjuk megismerni és megoldani a Custom Control készítésének legalapvetőbb problémáit és nehézségeit.

Egy Custom Control anatómiája

Custom Control létrehozására a következőképpen van lehetőség: Kattintsunk az egér jobb gombjával az adott projektre a Solution Explorerben, válasszuk az Add New Item opciót, és a felugró ablakban a Silverlight Templated Control sablont jelöljük ki (7-4 ábra)! A befejezéshez kattintsunk az Add gombra! Az eredmény egy mappa és két állomány.



7-4 ábra: **NumericUpDownControl** saját vezérlő felvétele

A **NumericUpDownControl.cs** fájlban található a vezérlőnk kódja. Közvetlenül a **Control** ősosztályból származik a vezérlőnk, és teljes mértékben egy hagyományos osztály kiinduláskor kapott kódjára emlékeztet, kivéve egyetlen extra sort. Ez a **Control.DefaultStyleKey** tulajdonságának beállítása egy **Type** objektumra, ami történetesen a **NumericUpDownControl** típusát reprezentálja. Mit jelent ez?

```
using System;
using System.Windows.Controls;

namespace MyControlsLibrary
{
    public class NumericUpDownControl : Control
    {
        public NumericUpDownControl()
        {
            this.DefaultStyleKey = typeof(NumericUpDownControl);
        }
    }
}
```

Többször felmerült már az a gondolat, hogy a vezérlő logikája és a kinézete teljesen szeparált, aminek következtében a vezérlők felfoghatók „lookless”-nek, azaz kinézet nélkülieknek. Persze így aligha lehetnének produktívak, ezért minden vezérlőhöz tartozik egy alapértelmezett kinézet. Ezt az alapértelmezett kinézetet határozza meg a **DefaultStyleKey** tulajdonság. Azt mondja a Silverlight-nak, hogy „menj és keress egy stílust, amit a **NumericUpDownControl** típushoz definiáltak”. Ezen a ponton kerül a képbe a másik állomány, a **Themes** mappában található **Generic.xaml**. Az alábbi kód mutatja ennek a fájlnak a tartalmát:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="clr-namespace:MyControlsLibrary">

  <Style TargetType="local:NumericUpDownControl">
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate TargetType="local:NumericUpDownControl">
          <Border Background="{TemplateBinding Background}"
                BorderBrush="{TemplateBinding BorderBrush}"
                BorderThickness="{TemplateBinding BorderThickness}">
          </Border>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</ResourceDictionary>
```

Nem túl nagy meglepetés, hogy ez a fájl valójában egy **ResourceDictionary**. Jól látható, hogy ennek az erőforrásfájlnak a tartalma egyetlen stílus, ami a **local:NumericUpDownControl** típushoz tartozik. A **local** prefix jelenleg a **MyControlsLibrary** névtérre mutat. A **DefaultStyleKey** tulajdonság értéke alapján épp ezt a stílust keressük. *Vagyis a **Generic.xaml**-ben található stílus írja le a Custom Controlunk alapértelmezett megjelenését!*

A **DefaultStyleKey** tulajdonság nem mindig a specifikus vezérlőre, néha annak egy ősére mutat. Ez sokszor teljes mértékben kielégítő, hiszen lehet, hogy az új vagy módosított vezérlő nem rendelkezik új vizuális megjelenéssel, csupán új funkcionalitással, így az őszvezérlőhöz tartozó alapértelmezett megjelenés tökéletes a leszármazott vezérlő számára.

A stílus definíció jelen pillanatban egyetlen üres **Border** objektumot definiál a **ControlTemplate** belsejében, így ezen a ponton még nem túlságosan izgalmas a vezérlőnk. Első lépés tehát a felhasználói felület kialakítása.

A **NumericUpDownControl** alapértelmezett megjelenése

A vezérlőnk egyszerű megjelenéssel fog rendelkezni, kialakításakor gondolunk a vezérlő átméretezésére is, így nem fix értékekkel, hanem arányokkal dolgozunk. Egy **TextBox** mellett két **RepeatButton**-t helyezünk el. Azért választottunk **RepeatButton**-t, hogy amennyiben a vezérlőnk gombjait nyomva tartjuk, úgy az érték a megfelelő irányba folyamatosan változzon. Ezt persze nem fogjuk követelményként definiálni, csupán az alapértelmezett sablon így fog működni.

```
<ControlTemplate TargetType="local:NumericUpDownControl">
  <Border Background="{TemplateBinding Background}"
        BorderBrush="{TemplateBinding BorderBrush}"
        BorderThickness="{TemplateBinding BorderThickness}">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
      </Grid.RowDefinitions>
```



```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="6*" />
    <ColumnDefinition Width="1*" />
</Grid.ColumnDefinitions>

<TextBox x:Name="txtValue" Grid.RowSpan="2" />
<RepeatButton x:Name="btnUp" Grid.Column="1" />
<RepeatButton x:Name="btnDown" Grid.Column="1" Grid.Row="1" />
</Grid>
</Border>
</ControlTemplate>
```

A Value Dependency Property implementálása

Az egyik első igényünk a **Value** tulajdonsággal kapcsolatban merült fel, ez pedig az adatkötés támogatása volt. A korábbi fejezetekben már többször is szerepeltek a *dependency property*-k. Ezek statikus tulajdonságok, melyek létrehozásuk pillanatában az adott vezérlő példány regisztrál. Legnagyobb előnyük, hogy ezek a tulajdonságok támogatják Silverlightban az adatkötést, a tulajdonság örökítést és az animációkat is. Ezekre a hagyományos tulajdonságok nem képesek. Éppen ezért a **Value** tulajdonságunkat is ilyen dependency propertyként fogjuk definiálni. A tulajdonság beregisztrálásakor a **PropertyMetadata** osztály konstruktorának átadható a tulajdonság alapértelmezett értéke, amit most nullában állapítunk meg.

Dependency Propertyt legegyszerűbben a *propdp* code snippet segítségével lehet készíteni.

```
public int Value
{
    get { return (int)GetValue(ValueProperty); }
    set { SetValue(ValueProperty, value); }
}

public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register("Value", typeof(int), typeof(NumericUpDownControl),
        new PropertyMetadata(0));
```

Azonban még mindig nem határoztuk meg, hogy ezt az információt hogyan és hol jelenítjük meg. A **TextBox** objektumunk **Text** tulajdonságát adatkötnünk kellene a **Value** tulajdonsághoz. Így ha valaki a **Value** tulajdonsághoz adatköt majd, és módosul az érték, akkor az automatikusan megjelenhet a **TextBox**-ban is. Felmerül azonban egy apró probléma. Nyilvánvaló, hogy a Binding-ot a **TextBox Text** tulajdonságán kell definiálni a **Generic.xaml**-ben, de hogyan fogunk magára a vezérlőre hivatkozni? A **Source** ezúttal nem egy statikus erőforrás vagy egy örökölt **DataContext**, hanem maga a vezérlő, aminek a sablonjában vagyunk. Segítségünkre siet a megoldásban a **RelativeSource** markup extension. Ennek a segítségével beállítható a **RelativeSourceMode** felsorolt típus **TemplatedParent** értéke. A **TemplatedParent** éppen arra a vezérlőre hivatkozik, aminek a sablonjában vagyunk. Az alábbi kódrészlet ezt a deklaratív adatkötést mutatja be:

```
<ControlTemplate TargetType="local:NumericUpDownControl">
    <Border Background="{TemplateBinding Background}"
        BorderBrush="{TemplateBinding BorderBrush}"
        BorderThickness="{TemplateBinding BorderThickness}">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="6*" />
                <ColumnDefinition Width="1*" />
```



```

        </Grid.ColumnDefinitions>
        <TextBox x:Name="txtValue" Grid.RowSpan="2" Text=
            "{Binding Path=Value, RelativeSource={RelativeSource TemplatedParent}}"/>
        <RepeatButton x:Name="btnUp" Grid.Column="1"/>
        <RepeatButton x:Name="btnDown" Grid.Column="1" Grid.Row="1"/>
    </Grid>
</Border>
</ControlTemplate>

```

A vezérlőnk ettől a ponttól a külvilág számára is adatköthetővé és használhatóvá vált. Az alábbi kódrészlet erre szolgál példaként, eredménye a 7-5 ábrán látható:

```

<my:NumericUpDownControl Name="numericUpDownControl1" Width="140" Height="25" />
<TextBox Height="25" Name="textBox1" Width="140"
    Text="{Binding Path=Value, Mode=TwoWay, ElementName=numericUpDownControl1}" />

```



7-5 ábra: TextBox adatkötve a NumericUpDownControlhoz

A legtöbb esetben, ha találkozunk egy vezérlővel, aminek van **Value** tulajdonsága és az változtatható is, akkor elvárjuk, hogy a vezérlő értesíteni tudjon minket erről a változásról. Így szükségessé válik egy kapcsolódó esemény — a **ValueChanged** — elkészítése is. Sőt egy kapcsolódó **EventArgs** osztályt is célszerű készítenünk, hogy tudjuk, mi volt az eredeti, és mi az újonnan beállított érték. A **ValueChanged** eseményt az alábbi módon definiáljuk:

```

public class NumericUpDownControl : Control
{
    public event EventHandler<ValueChangedEventArgs> ValueChanged;

    public NumericUpDownControl()
    {
        this.DefaultStyleKey = typeof(NumericUpDownControl);
    }
    //...
}

```

A kapcsolódó **ValueChangedEventArgs** definíciója:

```

public class ValueChangedEventArgs : EventArgs
{
    public int OldValue { get; set; }
    public int NewValue { get; set; }
}

```

Felmerül azonban egy fontos kérdés. Mikor kell ennek az eseménynek a bekövetkezését jelezni? Mikor változik meg a **Value** értéke? Sokan kapásból rávágnák, hogy természetesen, amikor a **Value** tulajdonság setter metódusa meghívódik. Ez azonban sajnos nem igaz. A **Value** tulajdonság itt gyakorlatilag csak egy közvetítő tulajdonság, a tényleges érték a vezérlő példányához bejegisztrált dependency property értéke. Annak ellenére, hogy az esetek 99%-ában, mi a **Value** tulajdonságon keresztül férünk a kérdéses **ValueProperty** elnevezésű dependency propertyhez, a Silverlight sajnos nem. Gondoljunk az animációkra, ott is arra, amikor kódból készítünk animációt. Soha egyetlen pillanatra sem merülne fel a

Value tulajdonság. Amit animálni próbálnánk, az a **ValueProperty** dependency property lenne. Ez azt jelenti, hogy a Silverlight tud úgy változni a **ValueProperty** tulajdonság értékén, hogy a **Value** tulajdonság setter metódusához hozzá sem érünk. Így a fenti ötletet máris elfelejthetjük.

Szerencsére van megoldás. A dependency propertyk létrehozásakor átadunk egy **PropertyMetadata** példányt, aminek a segítségével most az alapértelmezett 0 értéket állítjuk be. Ezen felül azonban egy **PropertyChangedCallback**-et is fogadhat. Ezt a callbacket fogja meghívni akkor, ha a dependency property értéke valóban megváltozik. Éppen erre van szükségünk, tehát nincs is más dolgunk, mint a callback metódust elkészíteni.

A callback függvény beregisztrálása a dependency propertyhez az alábbi kóddal történik:

```
public static readonly DependencyProperty ValueProperty =
    DependencyProperty.Register("Value", typeof(int), typeof(NumericUpDownControl),
        new PropertyMetadata(0,
            new PropertyChangedCallback(ValueChangedCallback)));
```

A callback függvényt a következő kód valósítja meg:

```
private static void ValueChangedCallback(DependencyObject sender,
    DependencyPropertyChangedEventArgs args)
{
    NumericUpDownControl control = sender as NumericUpDownControl;
    if (control == null) return;

    if (control.ValueChanged != null)
    {
        control.ValueChanged(control, new ValueChangedEventArgs
        {
            OldValue = (int)args.OldValue,
            NewValue = (int)args.NewValue
        });
    }
}
```

A dependency propertyk statikus jóságok, így a callback metódusnak is annak kell lennie. Ez persze statikus környezetet fog jelenteni számunkra. A metódus szignatúrája a következő:

```
static void ValueChangedCallback(DependencyObject, DependencyPropertyChangedEventArgs)
```

A **DependencyPropertyChangedEventArgs**-ban található két tulajdonság, az **OldValue** és a **NewValue**, amelyek a dependency property előző és az új értékét tartalmazzák. Nincs más dolgunk, mint egy **ValueChangedEventArgs** példányt készíteni belőlük, és a vezérlőn kiváltani a **ValueChanged** eseményt. A vezérlőre a referenciát a paraméterlistán található **DependencyObject** típusú paraméter (sender) típuskonverziójával szerezhetjük meg. Ezzel elkészült a vezérlőnk új változata, amely immáron képes a felhasználóját értesíteni a **Value** érték megváltozásáról is.

Eseménykezelő függvények bekötése

Ideje lenne életet lehelni gombjainkba is. Egy szimpla **UserControl** esetén, nem lenne min gondolkodnunk, feliratkoznánk XAML-ből a gombok **Click** eseményére. Itt ezt nyilvánvalóan nem tudjuk megtenni, hiszen ez az osztály nem a mögöttes kód, hanem a vezérlő maga, a **generic.xaml** pedig csak egy resource dictionary, arról nem is beszélve, hogy egyébként is **ControlTemplate**-ben vannak a gombjaink. Nincs más lehetőségünk, minthogy dinamikusan, kódból iratkozzunk fel gombjaink eseményére. A sablonban található vezérlőket **Name** tulajdonságuk alapján a **Control** **GetTemplateChild(string name)** metódusának segítségével szerezhetjük meg. A kérdés már csak az, hogy hol iratkozzunk fel? Erre a legalkalmasabb időpillanat a sablon alkalmazását követő lépés lenne. A **Control** osztály **OnApplyTemplate()** metódusát kell felüldefiniálnunk. Ezen a ponton biztosak lehetünk

abban, hogy a sablonban található vezérlőfa már elérhető. Az alábbi kódrészlet mutatja az **OnApplyTemplate** metódus felüldefiniálását:

```
public override void OnApplyTemplate()
{
    base.OnApplyTemplate();

    RepeatButton btnUp = this.GetTemplateChild("btnUp") as RepeatButton;
    btnUp.Click += (sender, args) => Value++;

    RepeatButton btnDown = this.GetTemplateChild("btnDown") as RepeatButton;
    btnDown.Click += (sender, args) => Value--;
}
```

A vezérlőnk most már működőképes. Lehet adatkötni, és a **btnUp** és a **btnDown** gombokra kattintva a vezérlő **Value** tulajdonsága változik pozitív illetve negatív irányba.

Vizuális állapotok a Custom Controlban

A következő igényünk az volt, hogy a vezérlő vizuális megjelenése tegyen különbséget a pozitív, illetve a negatív állapotok között. Ennek megfelelően készítenünk kell egy új **VisualStateGroup**-ot **SignStates** néven, és ebben pedig két új vizuális állapotot **PositiveState** és **NegativeState** néven. A **PositiveState**-hez olyan animációt kötünk, ami piros színbe billenti majd a **TextBox Foreground** tulajdonságát, a **NegativeState** esetén pedig kék színbe.

A módosított **ControlTemplate** így néz ki:

```
<ControlTemplate TargetType="local:NumericUpDownControl">
    <Border Background="{TemplateBinding Background}"
        BorderBrush="{TemplateBinding BorderBrush}"
        BorderThickness="{TemplateBinding BorderThickness}">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="SignStates">
                <VisualState x:Name="PositiveState">
                    <Storyboard>
                        <ColorAnimation To="Red"
                            Duration="00:00:00.3"
                            Storyboard.TargetName="txtValue"
                            Storyboard.TargetProperty=
                                "(Control.Foreground).(SolidColorBrush.Color)"/>
                    </Storyboard>
                </VisualState>
                <VisualState x:Name="NegativeState">
                    <Storyboard>
                        <ColorAnimation To="Blue"
                            Duration="00:00:00.3"
                            Storyboard.TargetName="txtValue"
                            Storyboard.TargetProperty=
                                "(Control.Foreground).(SolidColorBrush.Color)"/>
                    </Storyboard>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="6*" />
                <ColumnDefinition Width="1*" />
            </Grid.ColumnDefinitions>
        </Grid>
    </Border>
</ControlTemplate>
```

```
<TextBox x:Name="txtValue" Grid.RowSpan="2" Text=
    "{Binding Path=Value, RelativeSource={RelativeSource TemplatedParent}}"/>
<RepeatButton x:Name="btnUp" Grid.Column="1"/>
<RepeatButton x:Name="btnDown" Grid.Column="1" Grid.Row="1"/>
</Grid>
</Border>
</ControlTemplate>
```

A következő lépésben a vizuális állapotok közti váltást kell megoldanunk. Kódból ez a

```
VisualStateManager.GoToState(Control control, string stateName, bool useTransitions)
```

függvény segítségével lehetséges. A függvény első paramétere az a vezérlő, amelyben a vizuális állapotok definiálásra kerültek. Ez a mi esetünkben maga a **NumericUpDownControl**. A második paraméter a vizuális állapot neve, amelybe szeretnénk átmenni. A harmadik paraméter pedig egy **Boolean** érték, amivel jelezhetjük, hogy szeretnénk-e vizuális átmeneteket alkalmazni, vagy csupán átugrani kívánunk egyik állapotból a másikba. De hol váltsunk vizuális állapotot? Erre a legalkalmasabb hely a **ValueProperty** dependency propertyhez tartozó **ValueChangedCallback** függvény. Itt ugyanis mindig tudjuk, hogy milyen értékről milyen értékre változott a dependency propertynk. Figyeljünk, hogy itt statikus környezetben dolgozunk, így a **GoToState()** függvény első paraméterének a vezérlő példányának kell lennie. Ne aggódjunk amiatt, hogy ez a függvény állandóan meghívódik, ugyanis ha a vezérlőnk már abban az állapotban van, ahova menni akarunk, akkor ez a hívás semmit nem csinál, így ez nem okozhat problémát.

A **VisualStateManagement**-tel kiegészített callback függvény az alábbi:

```
private static void ValueChangedCallback(DependencyObject sender,
    DependencyPropertyChangedEventArgs args)
{
    NumericUpDownControl control = sender as NumericUpDownControl;
    if (control == null) return;
    if (control.ValueChanged != null)
    {
        control.ValueChanged(control, new ValueChangedEventArgs
        {
            OldValue = (int)args.OldValue,
            NewValue = (int)args.NewValue
        });
    }

    if ((int)args.NewValue > 0)
        VisualStateManager.GoToState(control, "PositiveState", true);
    else
        VisualStateManager.GoToState(control, "NegativeState", true);
}
```

Megjegyzés: Ez még így nem tökéletes. Induláskor ez a callback hamarabb fut le, minthogy a vizuális állapotok inicializálódtak volna. Ennek eredményeképpen induláskor fekete színű a TextBox **Foreground** tulajdonsága. Így szükséges még az **OnApplyTemplate()** metódus végén is ugyanezt a kódrészletet meghívni. Az egyszerűség és az átláthatóság kedvéért most ettől eltekintünk. Az eredményt a 7-6 ábra mutatja.



7-6 ábra: A **NumericUpDownControl** pozitív és negatív vizuális állapotban

Testreszabhatóság támogatása

Ezen a ponton kijelenthetjük, hogy a vezérlőnk a korábbi igényeknek megfelelően kész van. Rögtön neki is állhatunk felhasználni, és esetleg egy másik **ControlTemplate**-et készíthetünk hozzá. Azonban azt tapasztaljuk, hogy a vezérlőnk elszáll, amennyiben nem **btnUp** és **btnDown** elnevezésű **RepeatButton**-okat használunk, és a **Value** sem akar megjelenni az új **TextBox Text** tulajdonságában, hacsak nem vesszük fel kézzel a **Binding**-ot itt is. Igénylistánk utolsó pontja a testreszabhatóság támogatása volt, jelenlegi megoldásunkban ez azonban meglehetősen korlátozott. Kezdjük el a problémák megoldását!

Ahhoz, hogy tetszőleges nyomógomb típust támogatni tudjunk, az **OnApplyTemplate()** metódusban a **RepeatButton**-okhoz kapcsolódó műveleteket inkább a **RepeatButton** egy ősztyájával kell megvalósítanunk, mégpedig a **ButtonBase**-zel. Ebben az osztályban van definiálva a **Click** esemény, így ez nekünk bőven elegendő. A vezérlők jelenlétéhez nem fogunk ragaszkodni, így ha valaki nem akar a **NumericUpDownControl** vezérlőben „Up” irányt, akkor egyszerűen nem készít rá gombot. Így ha a sablonban nem találunk egy adott vezérlőt, akkor azt megfelelően kezeljük, és nem hagyjuk, hogy hibát okozzon az alkalmazásunkban.

A módosított — és így kevésbé szigorú — **OnApplyTemplate()** metódus az alábbi módon néz ki:

```
public override void OnApplyTemplate()
{
    base.OnApplyTemplate();

    ButtonBase btnUp = this.GetTemplateChild("btnUp") as ButtonBase;
    if(btnUp != null)
        btnUp.Click += (sender, args) => Value++;

    ButtonBase btnDown = this.GetTemplateChild("btnDown") as ButtonBase;
    if(btnDown != null)
        btnDown.Click += (sender, args) => Value--;
}
```

A következő probléma a **TextBox** adatkötése volt. Ez jelen pillanatban XAML-ből történik, azonban hogy egy később más által a **ControlTemplate**-be helyezett vezérlőn is működjön, így — akárcsak a gombok **Click** eseményeinek bekötését — ezt is dinamikusan, kódból, az **OnApplyTemplate()** függvényben kell elvégeznünk. Adatkötést a **Binding** osztály segítségével végezhetünk kódból.

A **TextBox** adatkötését az alábbi kóddal mozgathatjuk át az **OnApplyTemplate()**-be:

```
public override void OnApplyTemplate()
{
    base.OnApplyTemplate();

    ButtonBase btnUp = this.GetTemplateChild("btnUp") as ButtonBase;
    if(btnUp != null)
        btnUp.Click += (sender, args) => Value++;

    ButtonBase btnDown = this.GetTemplateChild("btnDown") as ButtonBase;
    if(btnDown != null)
        btnDown.Click += (sender, args) => Value--;

    TextBox txtValue = this.GetTemplateChild("txtValue") as TextBox;
    if (txtValue != null)
    {
        Binding binding = new Binding("Value");
        binding.Mode = BindingMode.TwoWay;
        binding.RelativeSource = new RelativeSource(RelativeSourceMode.TemplatedParent);
        txtValue.SetBinding(TextBox.TextProperty, binding);
    }
}
```

Természetesen így a XAML-ből eltávolíthatjuk a **Binding**-ot.

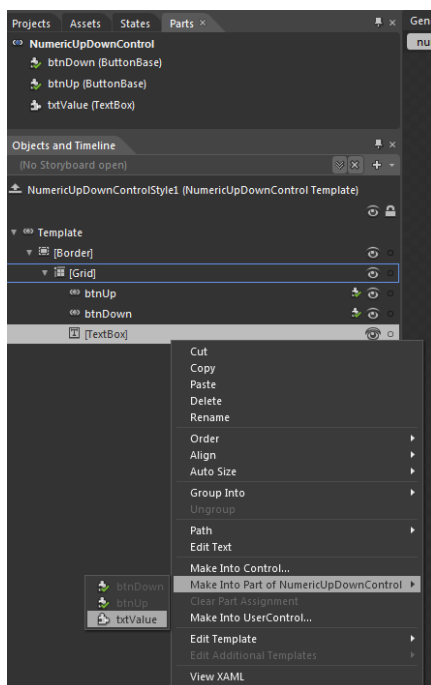
```
<ControlTemplate TargetType="local:NumericUpDownControl">
    <Border...
        ...
        <TextBox x:Name="txtValue" Grid.RowSpan="2"/>
        <RepeatButton x:Name="btnUp" Grid.Column="1"/>
        <RepeatButton x:Name="btnDown" Grid.Column="1" Grid.Row="1"/>
    </Grid>
</Border>
</ControlTemplate>
```

Ez a vezérlő most tetszőleges **ControlTemplate**-et tartalmazhat és nem fog elszállni. Ugyanakkor nem biztos, hogy tökéletesen fog működni. Egyetlenegy problémát nem sikerült megszüntetnünk, ez pedig a változónévtől való függés. Muszáj, hogy a vezérlőket **btnUp**, **btnDown** és **txtValue**-nak hívják, különben az **OnApplyTemplate**-ben nem tudunk hozzájuk férni. Ennyi kompromisszumot, érthető módon, sajnos kötnünk kell. Ez a kompromisszum azonban komoly korlátnak tűnik. A vezérlő későbbi felhasználója honnan tudhatná, hogy milyen elnevezésű és típusú elemeknek kell szerepelnie egy **ControlTemplate**-ben? Ez korábban nemcsak a Custom Controloknál, de bizony a standard Microsoftos vezérlőknél is okozott bőven problémát. Akkor a megoldás az volt, hogy meg kellett nézni az MSDN-en a vezérlők eredeti sablonját. Időközben azonban az Expression Blend okosabb lett. Jelenleg rendelkezésünkre áll egy attribútum, a neve: **TemplatePart**. Ennek az attribútumnak a segítségével tudunk jelezni a Blend felé, hogy ebben a vezérlőben vannak a funkciókhoz kapcsolódó kulcsfontosságú elemek. Az Expression Blend segítségével pedig könnyedén változtathatunk egy bármilyen egyszerű elemet az új sablonban **TemplatePart**-tá.

A tervezőeszköz (Blend) támogatását a **TemplatePart** attribútummal oldhatjuk meg:

```
namespace MyControlsLibrary
{
    [TemplatePart(Name = "btnUp", Type = typeof(ButtonBase))]
    [TemplatePart(Name = "btnDown", Type = typeof(ButtonBase))]
    [TemplatePart(Name = "txtValue", Type = typeof(TextBox))]
    public class NumericUpDownControl : Control
    {
        public event EventHandler<ValueChangedEventArgs> ValueChanged;
        //...
    }
}
```

Az Expression Blendben a Parts ablak jeleníti meg a felhasználható és szükséges **TemplatePart**okat, így a két gombunkat és a **TextBox**-ot. Ha új sablont készítünk és egy újonnan felvett elemünkre az egér jobb gombjával kattintunk, a Blend felkínál egy lehetőséget: „Make into Part of NumericUpDownControl”. A 7-7 ábrán jól látszik, hogy ezen belül kiválaszthatjuk, hogy a **TextBox** mely **TemplatePart** lesz. A kis „puzzle ikon a zöld pipával” azt jelenti, hogy az adott Part már a helyén van. A kis „puzzle ikon pipa” hiánya azt jelenti, hogy ehhez a Parthoz még nem rendeltünk semmit. A Parttá változtatás gyakorlatilag annyit jelent, hogy a megfelelő **Name** tulajdonságot felveszi a vezérlő. Ez a megközelítés jóval kényelmesebb, mint a dokumentációt olvasgatni.

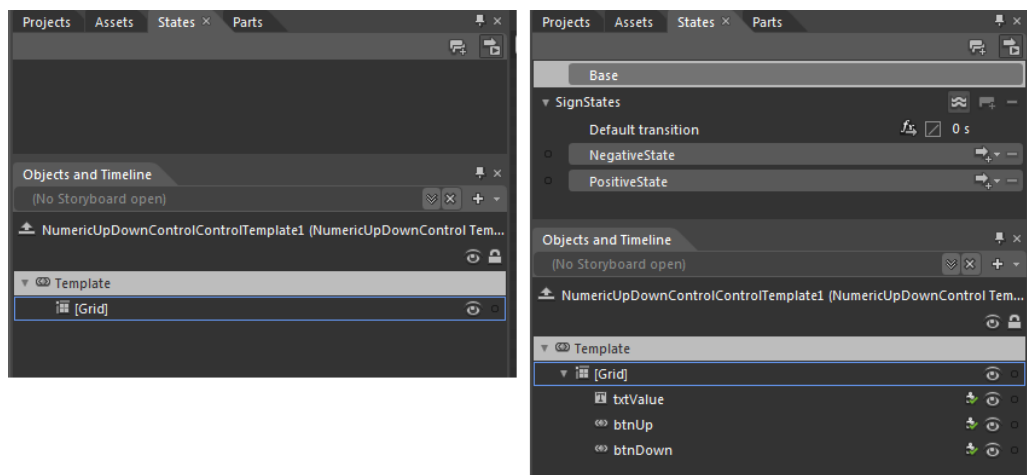


7-7 ábra: TemplatePart-ok bekötése az Expression Blendben

Ugyanez igaz a vizuális állapotokra is. Ha az új **ControlTemplate**-ben nincsenek definiálva vizuális állapotok **PositiveState** és **NegativeState** néven, akkor nem történik semmi a **Value** tulajdonság változásakor, legalábbis ami a megjelenést illeti. Amennyiben definiálunk ilyen állapotokat, akkor a hozzájuk kapcsolódó animációk lejátszódnak majd. Ismét abba a problémába ütközünk, hogy a vezérlő későbbi felhasználója nem tudhatja, milyen állapotok vannak egyáltalán. Korábban erre is a dokumentációk áttekintése (Microsoftos vezérlők esetén az MSDN tanulmányozása) volt a válasz. Szerencsére az Expression Blend ezen a téren is fejlődött. A **TemplateVisualState** attribútum jelenlétét vizsgálja az adott vezérlőn. Amennyiben megtalálja, akkor a megfelelő metaadatok alapján a States ablak tervezésidejű támogatást biztosít számunkra a vizuális állapotok kinézetének meghatározásához.

A vizuális állapotokhoz az alábbi módon definiálhatunk tervezésidejű támogatást:

```
namespace MyControlsLibrary
{
    [TemplatePart(Name = "btnUp", Type = typeof(ButtonBase))]
    [TemplatePart(Name = "btnDown", Type = typeof(ButtonBase))]
    [TemplatePart(Name = "txtValue", Type = typeof(TextBox))]
    [TemplateVisualState(GroupName="SignStates", Name="PositiveState")]
    [TemplateVisualState(GroupName = "SignStates", Name = "NegativeState")]
    public class NumericUpDownControl : Control
    {
        public event EventHandler<ValueChangedEventArgs> ValueChanged;
        // ...
    }
}
```



7-8 ábra: Blend Designer támogatás TemplateVisualState attribútum használatával és anélkül

Összefoglalás

A vezérlők készítése valóban a Silverlight alapú fejlesztés egyik legkomolyabb kihívása. Egyértelműen ez a terület igényli a legkomolyabb ismereteket a fejlesztőktől. Két típusú saját vezérlő készítésére van lehetőségünk: UserControl és Custom Control fejlesztésére. A UserControlok leggyakoribb használati területe a nézetek világa. A komplex felhasználói felület kisebb, önálló funkcionális egységekre való darabolásában játszanak fontos szerepet. Természetesen klasszikus értelemben vett, egyszerűbb vezérlők létrehozására is használhatók, de a testreszabhatóságnak komoly korlátokat szabnak. Custom Controlok segítségével a legegyszerűbb vezérlőktől a legkomplexxebbekig bármit készíthetünk, biztosítva az utólagos teljes testreszabhatóságot. A vezérlő megjelenése és a hozzá tartozó logika egymástól szinte teljesen elkülönül. A kettő közötti kapcsolatot lazán csatolva, konvenciók alapján biztosíthatjuk. Ennek kialakításában az Expression Blend komoly segítséget jelenthet.

8. Kommunikáció a kliens és a szerver között

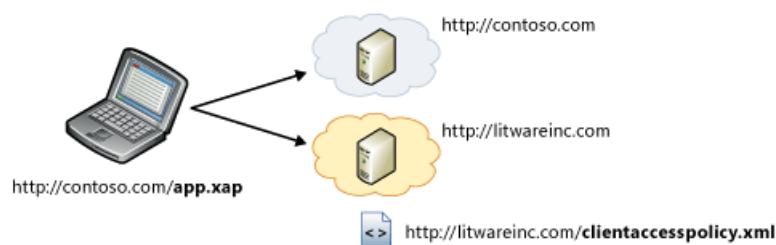
Ebben a fejezetben az adatkezelési feladatok *publikálás* és *adatelérés* részével fogunk foglalkozni. (Az adatkezelés egy másik kulcsfontosságú komponenséről, az *adatmegjelenítés*ről a 6. fejezetben olvashatsz bővebben.) Három technikát fogunk részletesen tárgyalni, másik hármat pedig csak érintőlegesen. Utunk során sokféle elgondolást és megvalósítást fogunk látni, de egy közös szemlélet mindegyikben fellelhető lesz: a kliens-szerver modell.

Kommunikáció a kliens-szerver modell alapján

A Silverlight egy kliensoldali technológia, ami azt jelenti, hogy a kódok a kliens számítógépén, annak a böngészőjében vagy akár azon kívül (OutOfBrowser mód esetén) hajtódnak végre. Ebből az következik, hogy nem töltődik le automatikusan az összes erőforrás az alkalmazást tartalmazó xap fájjal együtt, így valamilyen úton-módon a programnak futás közben kell dinamikusan elérnie a szerveren lévő erőforrásokat. Ilyen erőforrás lehet egy HD felbontású videó fájl vagy akár egy adatbázis is. Persze a szerverre nemcsak mint tárolóegységre tekinthetünk, hanem mint számítási kapacitásra is. Bonyolult, idő- és gépigényes műveletek kiszámítását ruházhatjuk át a kliensről a kiszolgálóra.

Általánosságban elmondható az, hogy a kiszolgáló az erőforrásait és a számítási kapacitását szolgáltatásokon keresztül ajánlja ki a külvilágnak. (Ha azok HTTP/HTTPS csatornán keresztül érhetőek el, akkor webszolgáltatásokról beszélünk.) Ha ezek a szolgáltatások *lazán kapcsolódó szabványos komponensek*, amelyek *felelősségi köre jól definiált*, és mindemellett *újra felhasználhatóak*, akkor az ilyen típusú rendszereket és a hozzájuk tartozó informatikai infrastruktúrát *Szolgáltatás Orientált Architektúrának* (SOA) szokás nevezni.

A szolgáltatásokat kínáló szervereket a Silverlight kapcsán két nagy csoportba oszthatjuk. Az egyikbe az a szerver tartozik, amelyen keresztül elérhető a Silverlightos alkalmazás, vagyis amelyik hosztolja azt. A másik csoportba tartozik az összes többi. A host szerverrel történő kommunikációhoz semmi extra dologra nincs szükségünk, míg a másik csoport tagjaival való kapcsolat felvételéhez bizonyos házirend (*policy*) fájlok megléte elengedhetetlen. (Ezekre azért van szükség, mert így a kiszolgáló tudja szűrni a bejövő kéréseket — domain vagy akár port alapján —, illetve szabályozni tudja a szolgáltatásokhoz kapcsolódók jogait.) Az ilyen típusú eléréseket hívják *Cross-Domain Access*-nek, amelyet a 8-1 ábra szemléltet.



8-1 ábra: Cross-Domain Access

A policy fájllokból kétfajta létezik, az egyik a Flashnél használt **crossdomain.xml**, a másik a pedig Silverlightnél újonnan bevezetett **clientaccesspolicy.xml**. Bármelyiket is választjuk, mindkét esetben a webkiszolgáló gyökérkönyvtárba kell elhelyeznünk az xml állományt. Ezekben hozzáférési jogokat lehet leírni, vagyis hogy kinek és mihez van joga, mint ahogyan azt a következő kód is mutatja (**clientaccesspolicy.xml**):

```
<?xml version="1.0" encoding="utf-8"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from>
        <domain uri="*" />
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

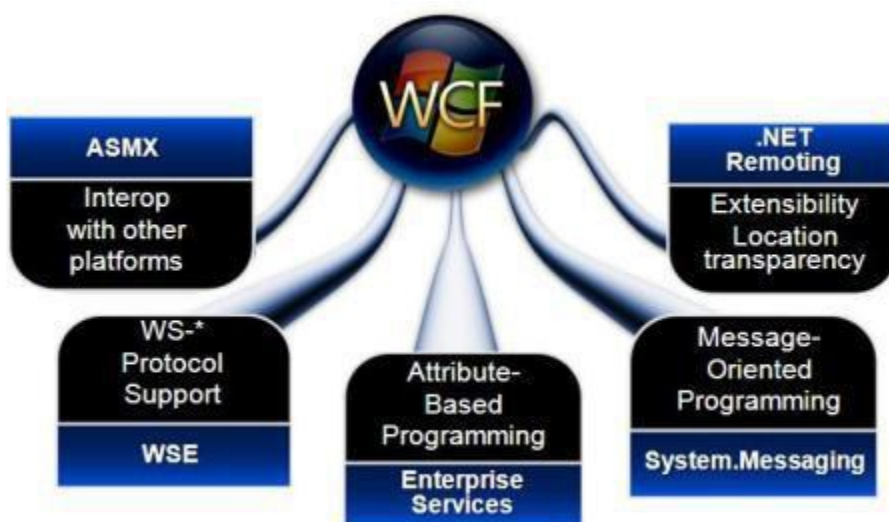
A Silverlight esetén érdemes még említést tenni a kliens-szerver modellen kívül a kliens-kliens közötti kommunikációról is. Ezzel a típussal az utolsó alfejezetben foglalkozunk bővebben.

WCF szolgáltatások készítése, elérése és használata

Történeti áttekintés, a WCF fejlesztési céljai

A *Windows Communication Foundation* a Windows operációs rendszer kommunikációs alrendszere, mely a .NET keretrendszer 3.0-ás verziójától kezdve érhető el a fejlesztők számára. A technológia kifejlesztésekor három fontos célt határoztak meg:

- Először is egy olyan új kommunikációs rendszert akartak megalkotni, mely elősegíti a szolgáltatásorientált alkalmazások fejlesztését. Az ilyen programoknál kétféle szerepkörrel lehet beszélni: a front-endről, illetve a back-endről. A front-end fogadja a felhasználótól az adatokat, kéréseket (ezt a szerepet a kliens szoftver tölti be), majd továbbítja ezt a hoszt felé, amely pedig meghívja a back-end-en a megfelelő feldolgozó metódust (ezt a szerepet a szolgáltatások töltik be).
- A következő cél az volt, hogy egy olyan rendszer lásson napvilágot, amely képes más típusú rendszerekkel is kommunikálni, nemcsak Windowsos alkalmazásokkal, idegen kifejezéssel élve az *interoperabilitásra* törekedtek. Például Java fejlesztők is tudják értelmezni a WCF szolgáltatások által generált választ. Ennek köszönhetően könnyen fejleszthetünk cross-platform alkalmazásokat is. Sőt a dolog meg is fordítható, vagyis olyan szituációk is elképzelhetők, ahol a szolgáltatásokat nem .NET környezetben implementálják, viszont .NET kliensek kapcsolódnak hozzájuk. Ehhez arra volt szükség, hogy a WCF standardizált formában küldje és fogadja a kéréseket, illetve a válaszokat (pl. SOAP, REST, stb.).
- A harmadik és egyben legfontosabb újítás, hogy egységes programozási modellt hoztak létre. A .NET 3.0 előtti időkben webes szolgáltatások írásakor WebService-eket kellett készíteni, melyek saját szintakszissal rendelkeztek (WebService, WebMethod, stb.). Nem webes szolgáltatások létrehozásakor a teljesítmény érdekében .NET Remotingot használtak a fejlesztők, melynek szintén sajátos implementációs módja volt. Ezt a szemléletet hivatott leváltani a WCF, mégpedig oly módon, hogy egységes programozási felületet és környezetet biztosít szolgáltatások készítéséhez, legyen szó akár egy biztonságos webszolgáltatásról, akár egy hatékony bináris formátumú üzenetsorról, esetleg egy peer-to-peer alapú kommunikációról. Ezt szemlélteti a 8-2 ábra.



8-2 ábra: Az egységes programozási felület és környezet

Ezen tulajdonságai mellett a WCF rendkívül rugalmas és testreszabható — beállítható például a generált válasz formátuma, a kommunikációs csatorna típusa, stb. —, illetve könnyen kiterjeszthető.

A WCF felépítése és az ABC

Minden egyes WCF szolgáltatás három részből áll: egy *Service osztályból*, egy *Host környezetből* és egy vagy több *végpontból*. Minden szolgáltatás a **System.ServiceModel** névtér **ServiceHost** osztályából származik, és egy interfészt valósít meg, amely leírja a szolgáltatás funkcionalitását, vagyis tartalmazza a kívülről elérhető metódusokat. A kiszolgáló oldali (host) futtató környezet információval szolgál például a különböző timeout-okról, a kommunikációs csatorna állapotáról, illetve a csatlakozott kliens(ek)ről. A végpontok pedig a kapcsolódási felületeket definiálják a kliensek és a szerver között. Ezeket egy-egy ABC (Address-Binding-Contract) hármassal szokás leírni.

A kliens és a kiszolgáló végpontjai között üzenet alapon történik a kommunikáció. A 8-3 ábra szemlélteti az üzenet alapú kommunikációt és a végpontok ABC felépítését.



8-3 ábra: Végpontok ABC szerkezete

- **Address** — ez a rész határozza meg, hogy a kliens milyen címen keresztül tudja elérni a kiszolgálót, illetve fordítva. A cím tartalmazza az IP-címet, a port számot, és adott esetben a szolgáltatás nevét is.
- **Binding** — a kommunikáció módját határozza meg (pl.: TCP/IP, Http), megszabja az adatátvitelt, a kódolást és a protokollt.
- **Contract** — a szolgáltatás interfészének a leírása, azt határozza meg, hogy milyen műveletekkel rendelkezik a szolgáltatás, és milyen adatobjektumokkal operál.

Silverlight esetén az elérhető Binding osztályok száma rendkívül korlátozott. Az alábbi négy használható csak: **BasicHttpBinding**, **NetTcpBinding**, **CustomBinding** és a **PollingDuplexHttpBinding**.

Szerződés (contract) típusokból ellenben jó pár elérhető, ezek közül a legfontosabbak:

- **ServiceContract**: egy olyan interfész vagy osztály, amelyet a WCF szolgáltatásnak meg kell valósítania. A kliens számára elérhető műveletek felületét definiálja.
- **OperationContract**: a **ServiceContract**-on belüli publikus műveleteket definiálja.
- **DataContract**: olyan adattípus, amelyet adatcserénél használ a rendszer.

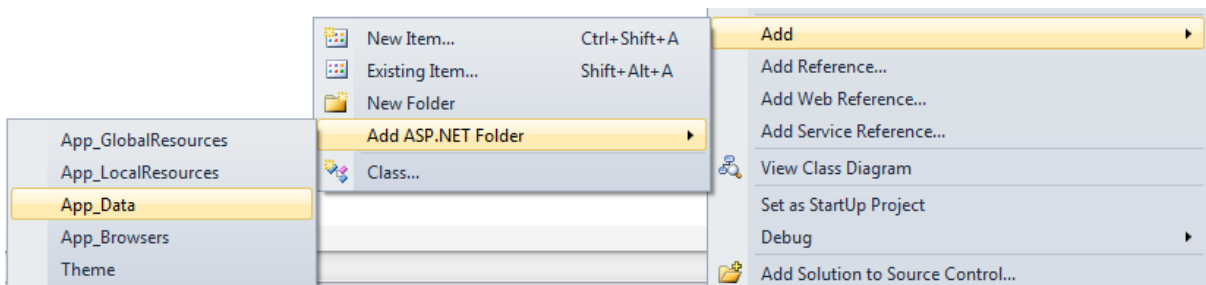
Szolgáltatások létrehozása

Silverlight esetén kétféleképpen is létre lehet hozni egy új WCF-es szolgáltatást. Az egyik lehetőség, hogy egy egyszerű *WCF Service*-t adunk az ASP.NET-es projekthez, a másik pedig az, hogy egy *Silverlight enabled WCF Service*-t. A kettő között az az alapvető különbség, hogy a Silverlightosnál nincs külön interfész a műveletek definiálásához, illetve alpból **customBinding**-ot használ (bináris kódolással) kommunikációs kötésnek. Mi ebben a fejezetben az egyszerűség kedvéért a Silverlight enabled WCF Service-es változatot fogjuk használni.

Készítsunk el egy egyszerű példaalkalmazást, mely egy WCF szolgáltatáson keresztül kínál lekérdezési lehetőségeket egy adatbázishoz! Az adatbázisunk legyen az *AdventureWorks Lite* példaadatbázis, melyet az alábbi címről lehet letölteni:

<http://msftdbprodsamples.codeplex.com/releases/view/37109#DownloadId=106391>.

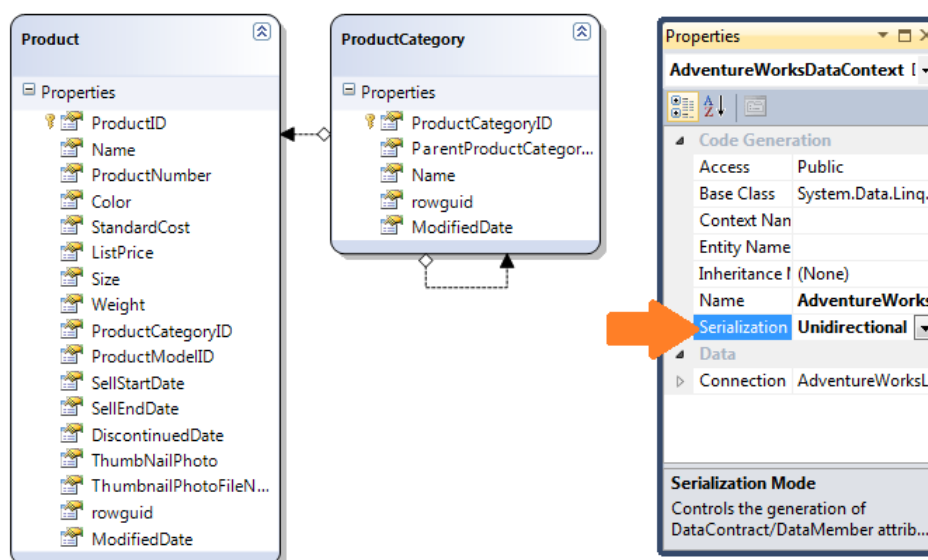
Hozzunk létre egy új Silverlight Application típusú alkalmazást **WCF_Demo** néven! Adjunk hozzá az ASP.NET-es projekthez egy új speciális mappát, az úgynevezett **App_Data** könyvtárat — ebben szokás tárolni az adatfájlokat. Jobb klikk a projekten >> Add >> Add ASP.NET folder >> App_Data, ahogyan azt a 8-4 ábra is mutatja.



8-4 ábra: *App_Data* könyvtár hozzáadása a projekthez

Ehhez a mappához adjuk hozzá az **AdventureWorksLT2008_Data.mdf** fájlt és a hozzá tartozó naplóállományt is.

Következő lépésként készítsük el a LINQ adatmodellt az adatbázishoz! Adjunk egy új *LINQ to SQL Classes* fájlt a projekthez **AdventureWorks.dbml** néven! Húzzuk a designer felületre a **Product** és a **ProductCategory** táblákat! Ahhoz, hogy egy WCF szolgáltatáson keresztül a Silverlightos klienseknek el tudjuk küldeni az entitásokat és bennük tárolt adatokat, arra van szükség, hogy a **DataContext Serialization Mode**-ját *None*-ről átállítsuk *Unidirectional*-re. A háttérben így az entitások el lesznek látva a megfelelő **DataContract** és **DataMember** attribútumokkal. (8-5 ábra)



8-5 ábra: LINQ adatmodell és a DataContext Serialization mode-ja

Adjunk a projektünkhöz egy új Silverlight enabled WCF Service osztályt, melynek az elemsablonját az Add New Item ablakban a Silverlight kategória alatt találhatjuk! Adjuk neki azt a nevet, hogy **ProductService.svc**! Ilyenkor valójában két fájl jön létre, egy **ProductService.svc** és egy **ProductService.svc.cs**. Az **svc** fájlon keresztül lehet majd elérni a szolgáltatást, az **svc.cs** fájlba pedig az implementációja kerül.

Nyissuk meg a **ProductService.svc.cs** fájlt, és töröljük ki a **DoWork** metódus kódját! Vegyünk fel egy privát adattagot a **DataContext**-ünkre, amely jelen esetben az **AdventureWorksDataContext** névre hallgat! A szolgáltatás konstruktorában inicializáljuk ezt a változót!

```
private AdventureWorksDataContext awdc;

public ProductService()
{
    awdc = new AdventureWorksDataContext();
}
```

Adjunk egy új metódust a szolgáltatáshoz, melyen keresztül lekérdezhethetjük a termékeket! Legyen a neve **GetProducts** és a visszatérési értéke pedig **List<Product>**! Maga az adatlekérés az alábbi módon néz ki:

```
[OperationContract]
public List<Product> GetProducts()
{
    List<Product> result = new List<Product>();
    var query = from p in awdc.Products
                select p;

    result = query.ToList();
    return result;
}
```

A kódrészlet legelső sora a legfontosabb, vagyis az **OperationContract**. Ha ezt az attribútumot lefelejtjük, akkor a metódus nem lesz elérhető kliensoldaltól. Készítsünk még egy metódust, ahol kategória alapján lehet szűrni a termékeket! Ennek a kódja az alábbi módon néz ki:

```
[OperationContract]
public List<Product> GetProductsByCategory(string category_name)
{
    List<Product> result = new List<Product>();
    var query = from p in awdc.Products
                where p.ProductCategory.Name.Contains(category_name)
                select p;

    result = query.ToList();
    return result;
}
```

Fontos: a metódusokat nem lehet túlterhelni!

Egy utolsó dolgot még meg kell tennünk ahhoz a szerveroldalon, hogy a kliensoldalról könnyebben használható legyen a szolgáltatás. Ehhez arra van szükségünk, hogy az alap **customBinding**-ot lecseréljük **basicHttpBinding**-ra. Ezt úgy tudjuk megtenni, hogy a **web.config**-ban megkeressük a **ProductService**-hez tartozó **services** elemet és ezen belül pedig az első **endpoint** taget. Ennek a **binding** attribútumát cseréljük le **basicHttpBinding**-ra, a **bindingConfiguration** attribútumát pedig töröljük ki! Íme, az átalakított service tag kódja:

```
<service name="WCF_Demo.Web.ProductService">
    <endpoint address="" binding="basicHttpBinding" contract="WCF_Demo.Web.ProductService"/>
    <endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />
</service>
```

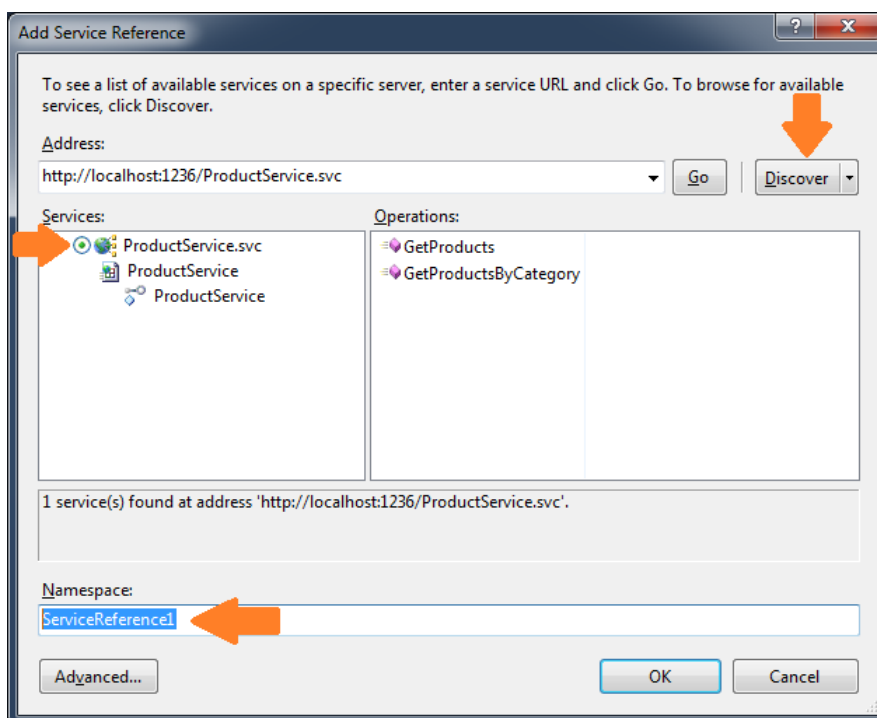
Ezzel el is készültünk a szolgáltatásunkkal. Ahhoz, hogy kliensoldalról használni is tudjuk, fordítsuk le a projektet!

Szolgáltatások elérése és használata

A szerveroldal után nézzük meg, mit kell ahhoz tennünk kliensoldalon, hogy használni tudjuk az előbb elkészített szolgáltatást! Először is egy referenciát kell rá felvennünk, melynek mellékhatásaként automatikusan generálódik egy proxy is.

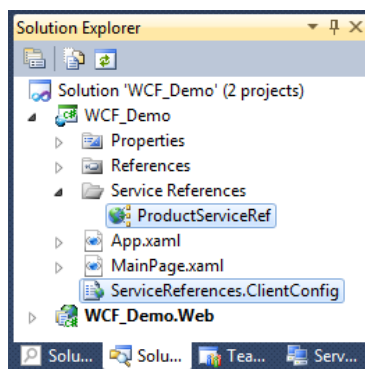
A proxy egy olyan osztály, mely tartalmaz egy referenciát a szolgáltatásra, és rajta keresztül lehet meghívni a szolgáltatás függvényeit. Vagyis egy olyan kliensoldali objektum, mely ugyanolyan felülettel rendelkezik, mint az általa helyettesített szolgáltatás.

Egy referencia felvétele a következőképpen történik: jobb klikk a *References* mappán vagy a Silverlight projekten, majd *Add Service Reference* menüpont. Itt a felugró ablakban kattintsunk a jobb oldalon található *Discover* gombra! Ennek hatására az *Address* legördülő lista, illetve az alatta lévő *Services* nézet feltöltődik adatokkal. Próbáljuk meg kinyitni a *ProductService.svc* node-ot, ha sikerül, akkor alatta megjelenik egy osztály, azon belül pedig egy interfész, amire kattintva a jobb oldali *Operations* lista is feltöltődik adatokkal. (8-6 ábra) Ha nem sikerülne kinyitni, akkor a *Services* nézet alatti részen található szövegben kattintsunk a linkre bővebb információért a hiba okáról!



8-6 ábra: Szolgáltatás-referencia felvétele

A dialógusablak alján a Namespace mezőben cseréljük le a névteret **ProductServiceRef**-re, majd kattintsunk az OK gombra! Ilyenkor a háttérben jó néhány új fájl létrejön, de mi ebből a Solution Explorerben csak egy új mappát és két új fájlt fogunk látni. (8-7 ábra)



8-7 ábra: Egy új mappa és két új fájl a Solution Explorerben

Ha megnézzük a **ServiceReferences.ClientConfig** fájl tartalmát, akkor azt láthatjuk, hogy bele van drótozva a szolgáltatás elérési útjának a portszáma. Ezzel az a probléma, hogy a gép minden egyes újraindítása után a Visual Studio más és más porton keresztül hosztolja az ASP.NET projektet. Ezért inkább dinamikusan mi rakjuk össze a szolgáltatás címét az alábbi módon:

```
string host = Application.Current.Host.Source.AbsoluteUri;
host = host.Substring(0, host.IndexOf("/ClientBin"));
string servicename = "ProductService";
string serviceurl = String.Format("{0}/{1}.svc", host, servicename);
```

Folytassuk tovább a fejlesztést a felhasználói felület elkészítésével! Szükségünk lesz egy TextBox-ra ahova be lehet írni a kategória nevét, egy gombra, amely meghívja a WCF szolgáltatás megfelelő metódusát, és egy DataGridView-re, amiben pedig meg lehet jeleníteni az adatokat. (Ha a beviteli mező üres, akkor az összes

termékre vagyunk kíváncsiak, ha viszont nem, akkor pedig a tartalma alapján szeretnénk szűrni őket.) Íme a **MainPage.xaml** kódja:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition />
    </Grid.RowDefinitions>
    <StackPanel Orientation="Horizontal">
        <TextBlock Text="Szűrési kategória alapján:" VerticalAlignment="Center"/>
        <TextBox x:Name="tb_categoryname" Width="100" />
        <Button x:Name="btn_getproducts" Content="Adatok lekérése"
            Click="btn_getproducts_Click" />
    </StackPanel>
    <sdk:DataGrid x:Name="dg_products" Grid.Row="1" />
</Grid>
```

A DataGrid vezérlőt a Toolboxról hozzuk be, ekkor ugyanis a Visual Studio automatikusan hozzáadja a projekthez hivatkozásként a megfelelő dll-t és beregisztrálja az sdk névteret is számunkra.

Menjünk át a **MainPage.xaml.cs** fájlba és a **using** taggel hivatkozzunk a **WCF_Demo.ProductServiceRef** névtérre, majd a következő kódrészletet helyezzük el a gomb eseménykezelő függvényében:

```
BasicHttpBinding binding = new BasicHttpBinding()
    { MaxReceivedMessageSize = int.MaxValue };
EndpointAddress address = new EndpointAddress(serviceurl);
ProductServiceClient client = new ProductServiceClient(binding, address);
if (tb_categoryname.Text.Length < 1)
{
    client.GetProductsCompleted += client_GetProductsCompleted;
    client.GetProductsAsync();
}
else
{
    client.GetProductsByCategoryCompleted += client_GetProductsByCategoryCompleted;
    client.GetProductsByCategoryAsync(tb_categoryname.Text);
}
```

A proxyn keresztül aszinkron módon tudjuk meghívni a metódusokat. (Az aszinkronitás egy kulcsfontosságú dolog a RIA alkalmazásoknál, ezért a szolgáltatások elérése sem maradhat ki ebből a mókából.) A szerveren definiált függvények itt egy metódus és egy esemény formájában vannak jelen, ahol az aszinkronitás miatt a függvénynek (*Async) nincs visszatérési értéke, ugyanis azt majd az eseményt lekezelő függvényben (*Completed) lehet elérni.

WCF-nél alapból a válasz mérete maximum 64 KByte lehet, de mivel a **Product** tábla képeket is tartalmaz bináris formában, ezért a **MaxReceivedMessageSize** tulajdonság értékét meg kell növelni.

Már csak a két **Completed** eseménykezelő függvény törzsének megírása van hátra. A WCF-es aszinkron metódusok **EventArgs** objektuma kicsit más, mint amit eddig megszokhattunk a Silverlightban. Két fontos tulajdonsága van, az **Error** és a **Result**. Az **Error** információt tartalmaz arról, hogy történt-e hiba a szerver oldalon (a hiba okáról nem!). A **Result** a visszatérési értéke a függvényeknek. A **Completed** eseményekhez rendelt kód emiatt általában úgy néz ki, hogy megvizsgáljuk először, hogy történt-e hiba, ha nem, akkor feldolgozzuk a kapott eredményt, ha viszont igen, akkor azt jelezzük a felhasználónak. Íme, az eseménykezelő függvények kódja:


```

void client_GetProductsCompleted(object sender, GetProductsCompletedEventArgs e)
{
    if (e.Error == null)
        dg_products.ItemsSource = e.Result;
    else
        MessageBox.Show("Hiba történt az adatok lekérdezése közben!");
}

void client_GetProductsByCategoryCompleted(object sender,
    GetProductsByCategoryCompletedEventArgs e)
{
    if (e.Error == null)
        dg_products.ItemsSource = e.Result;
    else
        MessageBox.Show("Hiba történt az adatok lekérdezése közben!");
}

```

Utolsó lépésként futtassuk az alkalmazást! (8-8 ábra)

Szűrési kategória alapján:	Mount	Adatok lekérése				
ProductID	Name	ProductCategoryID ▲	ProductNumber	Color	StandardCost	ListP
771	Mountain-100 Silver, 38	5	BK-M82S-38	Silver	1912.1544	3399
772	Mountain-100 Silver, 42	5	BK-M82S-42	Silver	1912.1544	3399
773	Mountain-100 Silver, 44	5	BK-M82S-44	Silver	1912.1544	3399
774	Mountain-100 Silver, 48	5	BK-M82S-48	Silver	1912.1544	3399
775	Mountain-100 Black, 38	5	BK-M82B-38	Black	1898.0944	3374
776	Mountain-100 Black, 42	5	BK-M82B-42	Black	1898.0944	3374
777	Mountain-100 Black, 44	5	BK-M82B-44	Black	1898.0944	3374
778	Mountain-100 Black, 48	5	BK-M82B-48	Black	1898.0944	3374

8-8 ábra: „Mount” kifejezést tartalmazó kategóriákra szűrt termékek

Data Transfer Object használata

Az előző példaalkalmazással van egy kis probléma: sok felesleges adat utazik az éterben. Amikor adatbázisokkal dolgozunk, nagyon ritkán van szükségünk a rekordok összes mezőjére. Általában csak a mezőknek egy részhalmazával dolgozunk. Ezért jó, hogy van Anonymus típus, ugyanis így a select utasításnál nem kell a teljes entitást lekérnünk, hanem megadhatjuk annak egy tetszőleges projekcióját.

```

select new
{
    ProductId = p.ProductID,
    ProductName = p.Name,
    CategoryName = p.ProductCategory.Name,
    ...
}

```

Oké, de akkor mi legyen a függvények visszatérési értéke? Minden LINQ lekérdezés egy **IQueryable<T>**-t ad alapból vissza, viszont anonymus típus esetén a T-t nem ismerjük (emiat a **ToList()** sem hívható meg rajta), ezért a függvények visszatérési értéke simán **IQueryable** kell, hogy legyen. Tegyük fel, hogy átírtuk a szolgáltatást, frissítettük a kliensoldalon a proxyt, ekkor azt látjuk, hogy a Result típusa object. Sebaj, castoljuk. Minden szép és jó, fordít, futtat ... kattint ... *CommunicationException*. Indoklás: *The remote server returned an error: NotFound*, ezt kapjuk a **GetProducts** meghívásakor. A probléma, ami miatt nem működik ez a módszer, hogy az anonymus típusok nincsenek ellátva a WCF-es adatobjektumokhoz szükséges attribútumokkal, emiatt a szolgáltatás nem tudja őket átküldeni a hálózaton.

Létezik egy másik módszer, amely képes megoldani ezt a problémát: ez a *Data Transfer Object* (DTO). Mint ahogyan az a nevéből is adódik, ezeket az objektumokat arra használjuk, hogy adatokat juttassunk el

bennük a szolgáltatástól a kliensekhez. Ahhoz, hogy a WCF ezeket az objektumokat megfelelőképpen tudja sorosítani (szerializálni) a hálózaton keresztül történő utaztatáshoz, arra van szükség, hogy ellássuk az osztályt egy **DataContract** attribútummal és minden tulajdonságát egy **DataMember**-rel. (Vagyis a dinamikusan generált anonyim típusok helyett statikusan létre kell hoznunk olyan osztályokat, melyeket a WCF már képes használni.)

Hozzunk létre egy DTO nevezetű mappát az ASP.NET-es projekten belül, és adjunk hozzá egy új osztályt **Product** néven! Az alábbi kódot helyezzük el a fájlba:

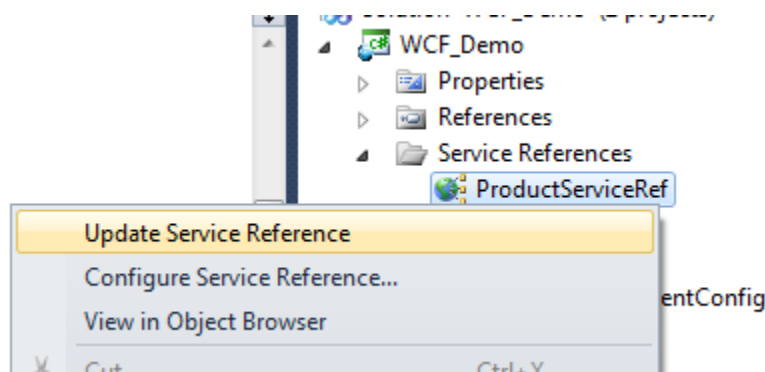
```
using System.Runtime.Serialization;

namespace WCF_Demo.Web.DTO
{
    [DataContract]
    public class Product
    {
        [DataMember]
        public int ProductId { get; set; }
        [DataMember]
        public string ProductName { get; set; }
        [DataMember]
        public string CategoryName { get; set; }
        [DataMember]
        public decimal ListPrice { get; set; }
        [DataMember]
        public decimal StandardCost { get; set; }
        [DataMember]
        public string Size { get; set; }
        [DataMember]
        public decimal? Weight { get; set; }
    }
}
```

Ezek után cseréljük le a **ProductService**-en belül a függvények visszatérési értéket és a result változók típusát **List<Product>**-ről **List<DTO.Product>**-ra, illetve a LINQ lekérdezések **select p** részét az alábbi kódrészletre:

```
select new DTO.Product
{
    ProductId = p.ProductID,
    ProductName = p.Name,
    CategoryName = p.ProductCategory.Name,
    ListPrice = p.ListPrice,
    StandardCost = p.StandardCost,
    Size = p.Size,
    Weight = p.Weight
}
```

Fordítsuk le az ASP.NET projektet! Ezek után frissítsük a szolgáltatás referenciát a Silverlight alkalmazásnál, a 8-9 ábrán látható módon.



8-9 ábra: Szolgáltatás referencia frissítése

Az alkalmazást futtatva a 8-10 ábrán látható eredményt kapjuk.

Szűrési kategória alapján: Mount		Adatok lekérése					
ProductId	ProductName	CategoryName	ListPrice	StandardCost	Size	Weight	
739	HL Mountain Frame - Silver, 42	Mountain Frames	1364.5000	747.2002	42	1233.76	
740	HL Mountain Frame - Silver, 44	Mountain Frames	1364.5000	706.8110	44	1251.91	
741	HL Mountain Frame - Silver, 48	Mountain Frames	1364.5000	706.8110	48	1270.05	
742	HL Mountain Frame - Silver, 46	Mountain Frames	1364.5000	747.2002	46	1288.20	
743	HL Mountain Frame - Black, 42	Mountain Frames	1349.6000	739.0410	42	1233.76	

8-10 ábra: Silverlight kliens DTO használatával

Egy WCF szolgáltatás esetén a hibák kezelése, a normál .NET-nél megszokotthoz képest, eltérően működik, ezért erősen ajánlott az alábbi oldalon található videó megtekintése és a lap alján lévő linkek felkeresése: <http://channel9.msdn.com/Shows/SilverlightTV/Silverlight-TV-46-Whats-Wrong-with-my-WCF-Service>.

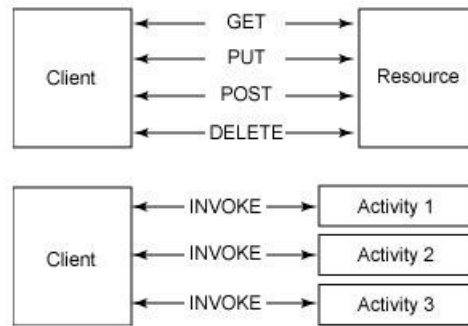
WCF Data Services, avagy az adatok egyszerű kipublikása

Amikor adatbázisokkal dolgozunk, akkor általában úgynevezett **CRUD** típusú műveleteket hajtunk végre. A CRUD a *Create*, *Read* (vagy *Retrieve*), *Update*, *Delete* szavakból kreált mozaikszó, vagyis a létrehozás, lekérdezés, frissítés és törlés műveletek gyűjtőneve. Ezeket a műveleteket ismételjük unos-untalan az adatelérési rétegben (a későbbiekben csak *Data Access Layer*, azaz *DAL*). A WCF Data Services ezen a sok mechanikus munkát igénylő helyzeten javít oly módon, hogy 3-4 sornyi kóddal ki tudunk publikálni teljes adattáblákat egy speciális, de szabványos formátumban. Ez a formátum az úgynevezett REST.

A REST protokoll és az OData lekérdezőnyelv

A REST a *REpresentational State Transfer* szavakból kreált mozaikszó, mely egy architektúráis stílus elosztott hipermédia rendszerekhez. (A REST nem hivatalos szabvány. Annak ellenére, hogy a REST maga nem szabvány, szabványos technológiákat használ a működése során, mint például a HTTP, URL, HTML, text/xml, image/jpeg.) Vagyis a SOAP-hoz hasonlóan, ez is arra szolgál, hogy a kliens és a szerver közötti kommunikáció szabványos formában történjen. A SOAP és a REST között alapvető koncepcióbeli különbségek vannak.

A REST-nél az URL-be kódolva erőforrás azonosítók segítségével kommunikálunk a kiszolgálóval (*resource-oriented*), míg a SOAP-nál függvényhívásokat intézünk a szolgáltatás felé (*activity-oriented*). A két protokoll közötti alapvető szemléletbeli különbséget szemlélteti a 8-11 ábra.



8-11 ábra: REST (felső) és SOAP (alsó) kommunikáció összehasonlítása

Íme, egy egyszerű példa egy SOAP és egy REST kérésre, melyek a Mountain Bikes kategóriába tartozó termékeket kérik le.

SOAP kérés:

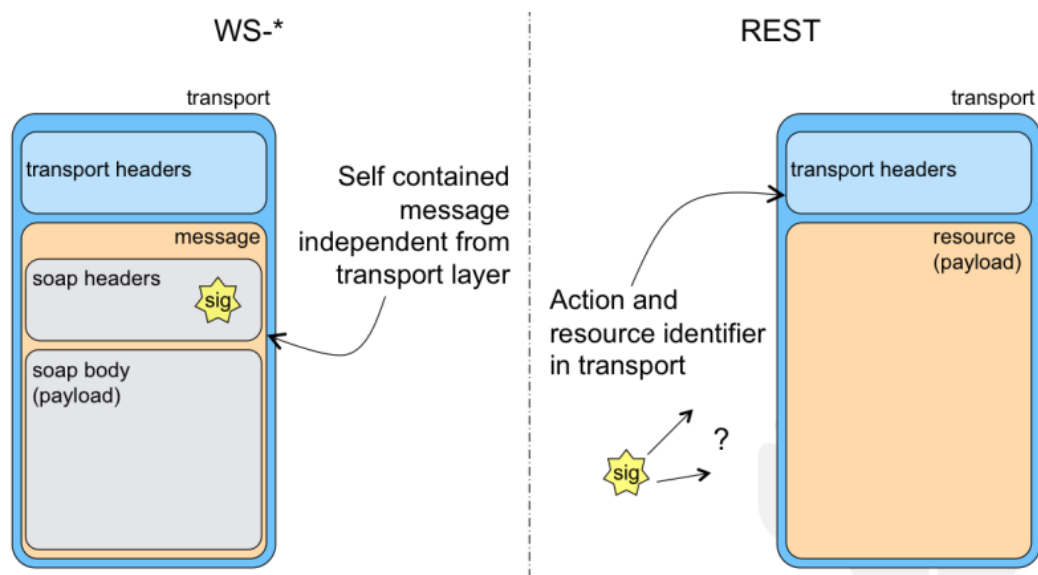
```
POST http://127.0.0.1:1236/ProductService.svc HTTP/1.1
Accept: */*
Referer: http://ip4.fiddler:1236/ClientBin/WCF_Demo.xap
Accept-Language: hu-HU
Content-Length: 177
Content-Type: text/xml; charset=utf-8
SOAPAction: "urn:ProductService/GetProductsByCategory"
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; ...
Host: 127.0.0.1:1236
Connection: Keep-Alive
Pragma: no-cache

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <GetProductsByCategory>
      <category_name>Mountain Bikes</category_name>
    </GetProductsByCategory>
  </s:Body>
</s:Envelope>
```

REST kérés:

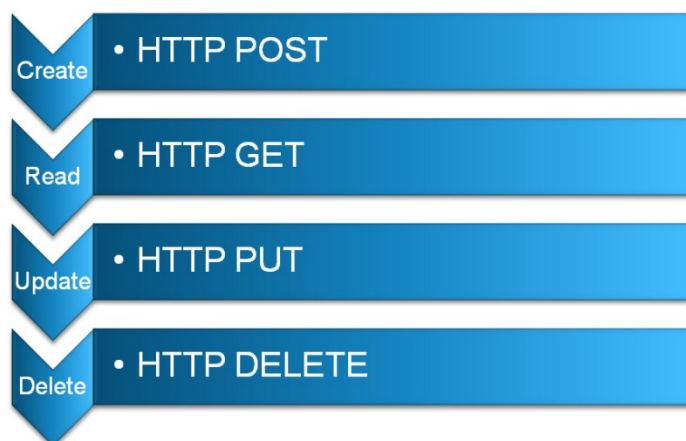
```
GET http://127.0.0.1:10834/ProductService.svc/Products?$filter=ProductCategoryID%20eq%205 HTTP/1.1
Accept: application/x-ms-application, image/jpeg, application/xaml+xml, image/gif, image/pjpeg, application/x-ms-xbap, ..., */*
Accept-Language: hu
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; ...
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Host: 127.0.0.1:10834
```

Mint ahogyan az a fentebbi kódrészletekből is látszik, a REST-es üzenetek tömörebbek és rövidebbek is, mint a SOAP-osak. Ezt a tényt a 8-12 ábra talán még jobban szemlélteti.



8-12 ábra: SOAP és REST üzenetek formátumának összehasonlítása

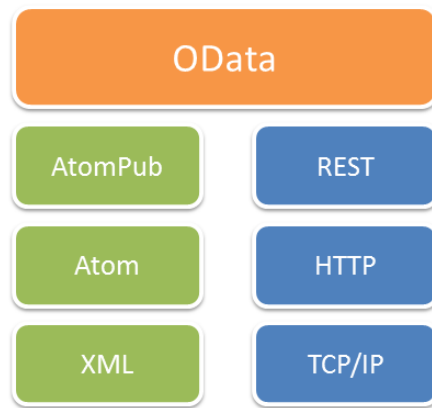
A tömörség annak is köszönhető, hogy a REST-nél minden egyes CRUD műveletnek megvan a neki megfelelő HTTP utasítás (verb), amint azt a 8-13 ábra is szemlélteti. Így már a kérés módjából kiderül, hogy mit akarunk kezdeni az adott erőforrással.



8-13 ábra: CRUD műveletek HTTP utasítás megfelelői

Egy szolgáltatást akkor nevezzünk *RESTful Service*-nek, ha egyértelműen azonosíthatók az erőforrások, manipulációjuk a reprezentációjukon keresztül történik, és önleíró üzeneteket használ.

A CRUD típusú műveletek közül az egyik leggyakrabban használt kétségtelenül a Read. Ezért ehhez egy külön lekérdező nyelvet is megalkottak, az úgynevezett OData-t, amely az *Open Data Protocol* rövidítése. (Az OData-n keresztül adatmódosítás is lehetséges, de mi ezzel a részével nem foglalkozunk.)



8-14 ábra: Bal oldalt az adatcserélési formátumok verme, jobb oldalt pedig a kommunikációs formáké

Maga az OData eléggé sok mindent definiál, ezek közül minket most csak a műveletek érdekelnek. Az egyik legalapvetőbb a *filter*, mellyel szűrési feltételt szabhatunk meg, íme, egy egyszerű példa:

```
localhost/ProductService.svc/Products?$filter=ProductID eq 1
```

Látható, hogy először az adattáblát (erőforrást) kell megadni, utána pedig GET paraméterként a szűrési feltételt, mely a **\$filter=** kulcsszóval kezdődik. Mivel itt csak egyetlenegy elemre vagyunk kíváncsiak, ezért a fenti példa ekvivalens az alábbival:

```
localhost/ProductService.svc/Products(1)
```

Gyakran nem a teljes objektumra van szükségünk, hanem csak egy részére, ilyenkor a *select* művelettel tudunk projektálni. Íme, egy példa:

```
localhost/ProductCategories(1)?$select=CategoryId,Name
```

Egy szintén gyakran előforduló művelet az *expand*, mellyel a relációban lévő rekordokat tudjuk összekapcsolni, vagyis például az 1-több kapcsolatnál az 1 oldalon lévő entitást ki tudjuk bővíteni a hozzá kapcsolódó „több” oldali megfelelő rekordokkal. Íme, egy példa:

```
localhost/ProductCategories(1)?$expand=Products
```

A *WCF Data Services-es szolgáltatások létrehozása* című részben mindezt megnézzük majd működés közben is.

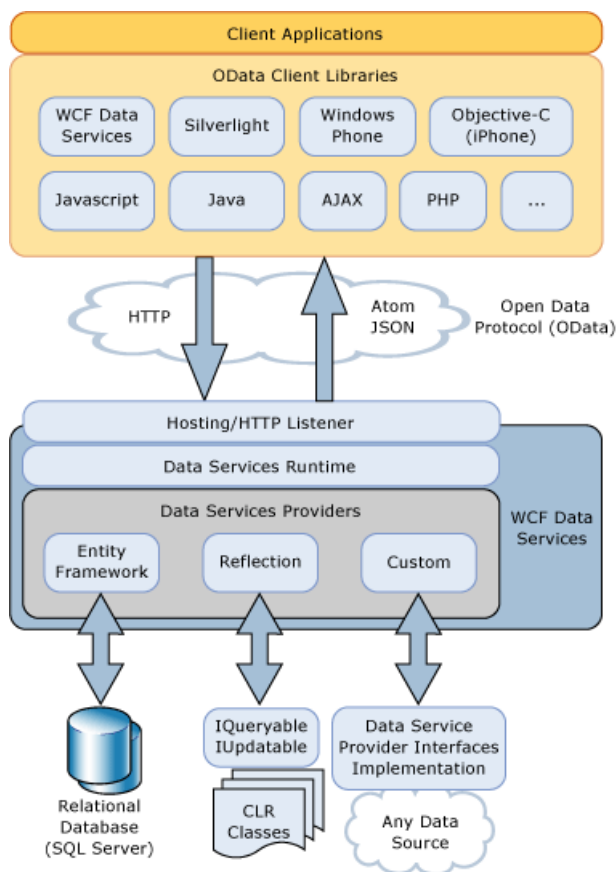
WCF Data Services alapok

A WCF Data Services egy keretrendszer, mely *adatcentrikus* szolgáltatások létrehozását és használatát megkönnyítő minták és könyvtárak kombinációjából épül fel.

Nézzük meg először a szerveroldalt! Amikor létrehozunk egy ilyen típusú szolgáltatást, akkor meg kell adnunk egy *Entitás adatmodellt*, illetve meg kell szabnunk azt is, hogy mely adattáblákat szeretnénk kipublikálni és milyen módon (csak olvasható, esetleg szerkeszthető is). Ezek mellett lehetőségünk van saját műveletek létrehozására is (például egyedi több paraméteres lekérdezések), illetve a háttérben automatikusan generált kódokat is felül lehet definiálni (például extra validációval kibővíteni a beszúrást). Utóbbiakat *Interceptor*-nak hívják a WCF Data Services terminológiában.

Térjünk át a kliensoldal részletezésére! A WCF Data Services könyvtárai oly módon leegyszerűsítik a szolgáltatások használatát, hogy egyszerűen csak egy LINQ lekérdezést kell megfogalmaznunk a

kliensoldalon. A háttérben a rendszer ezt átalakítja egy OData lekérdezésre, majd továbbítja ezt a szervernek, és a REST válaszban lévő ATOM vagy JSON formátumban tárolt adatot automatikusan betölti .NET-es objektumokba. A rendszer architektúra a 8-15 ábrán látható.



8-15 ábra: WCF Data Services szerkezete, felépítése

Szolgáltatások létrehozása

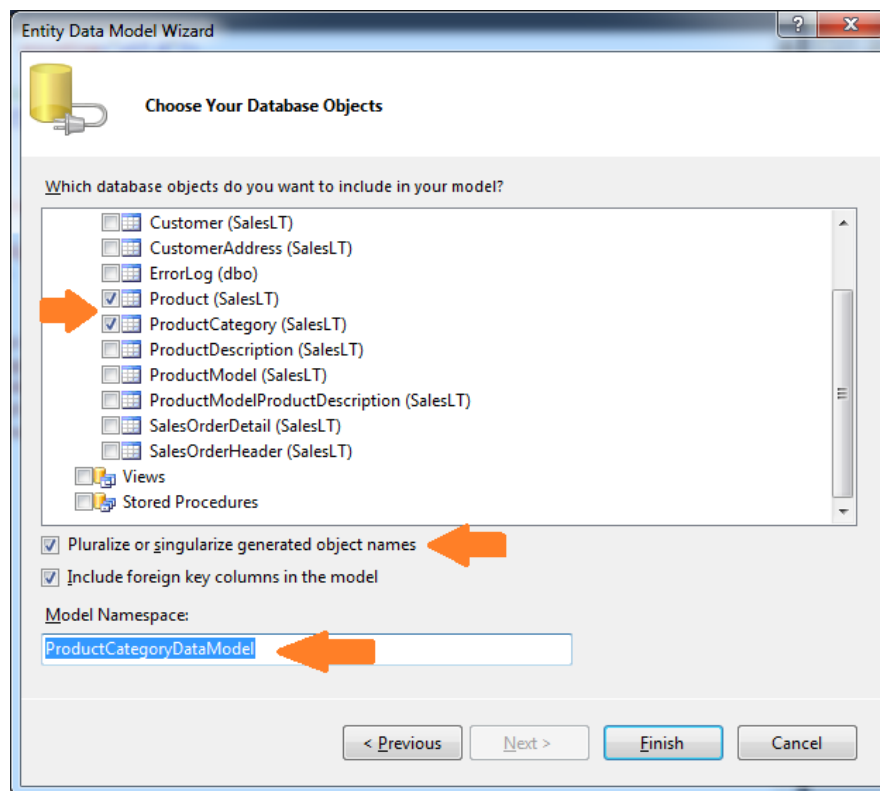
Egy WCF Data Services szolgáltatás kódolás szempontjából nem más, mint egy némileg átalakított WCF-es szolgáltatás. A legfontosabb különbség, hogy itt a szolgáltatás nem a **ServiceHost** osztályból van származtatva, hanem a **System.Data.Services** névtér **DataService<T>** osztályából, ahol a **T** egy **EntityDataModel** típust reprezentál. Első lépésként tehát egy adatmodellt kell elkészítenünk.

Hozzunk létre egy új projektet **WCFDataServices_Demo** néven, és az előző alfejezetben láttak alapján húzzuk be az AdventureWorks Lite adatbázist az ASP.NET-es projektbe!

Adjunk az alkalmazásunkhoz egy új **ADO.NET Entity Data Model** fájlt **ProductCategory.edmx** néven!

Adatbázisból generáltassuk a modellt és az eltárolandó ConnectionStringet nevezzük át

ProductCategory_DataEntities-re! Válasszuk ki a **Product**, illetve **ProductCategory** táblákat! Alul kattintsuk be a *Pluralize or singularize generated objects names* opciót, és a Modell névterének állítsuk be a **ProductCategoryDataModel**-t, ahogyan azt a 8-16 ábra is mutatja!



8-16 ábra: Entity Adatmodell elkészítése

Adjunk a projektünkhöz **ProductService.svc** néven egy új *WCF Data Service* fájlt, amelynek az elemsablonja az Add New Item dialógus Web kategóriájában található! Az osztály definíció legelső sorában lévő `/* TODO: put your data source class name here */` részt cseréljük le **ProductCategory_DataEntities**-re, vagyis ezt a sort kell, hogy kapjuk:

```
public class ProductService : DataService<ProductCategory_DataEntities>
```

Az **InitializeService** metóduson belül helyezzük el az alábbi két sort:

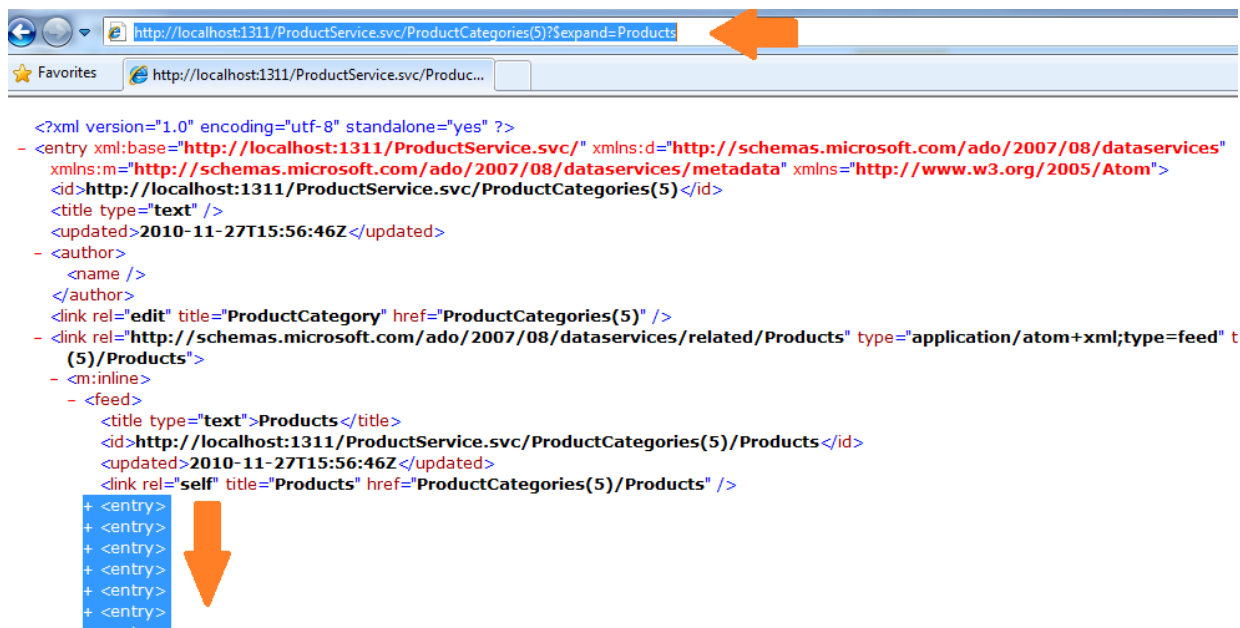
```
config.SetEntitySetAccessRule("Products", EntitySetRights.AllRead);  
config.SetEntitySetAccessRule("ProductCategories", EntitySetRights.AllRead);
```

Ezekkel engedélyeztük a hozzáférést (csak olvasásra) a termékek és termékkategóriák táblákhoz. Kattintsunk jobb egérgombbal a Solution Explorerben a **ProductService.svc** fájlra, és a helyi menüből válasszuk ki a *View in Browser* parancsot! A böngészőben a 8-17 ábrán látható tartalmat kell, hogy lássuk.

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>  
- <service xml:base="http://localhost:1311/ProductService.svc/" xmlns="http://www.w3.org/2007/app">  
  <workspace>  
    <atom:title>Default</atom:title>  
    - <collection href="Products">  
      <atom:title>Products</atom:title>  
    </collection>  
    - <collection href="ProductCategories">  
      <atom:title>ProductCategories</atom:title>  
    </collection>  
  </workspace>  
</service>
```

8-17 ábra: Az adattáblák nevei egy REST-es üzenetben Atom formátumban

A cím végére biggyesszük oda az alábbi szöveget: `/ProductCategories(5)?$expand=Products`, majd vizsgáljuk meg a kapott eredményt (lásd 8-18 ábra)! Az **expand** segítségével az 5. termékkategóriába tartozó termékek belekerültek a termékkategória leírásába.



8-18 ábra: Az 5. termékkategória és a hozzá tartozó termékek

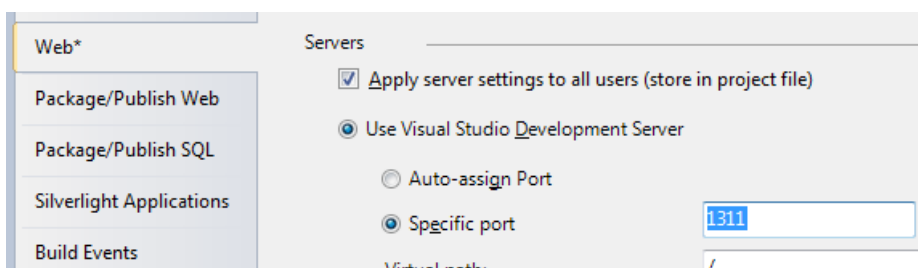
Ha az **expand** részt elhagyjuk, akkor nem fognak automatikusan belekerülni a termékek az eredményhalmazba, ugyanis az Entity DataModel úgynevezett *lazy loading*-ot (lusta betöltést) használ, így csak akkor tölti be a kapcsolódó rekordokat, ha explicit módon megkérjük rá.

Ezzel el is készültünk az adatok publikálásával.

Szolgáltatások elérése és használata

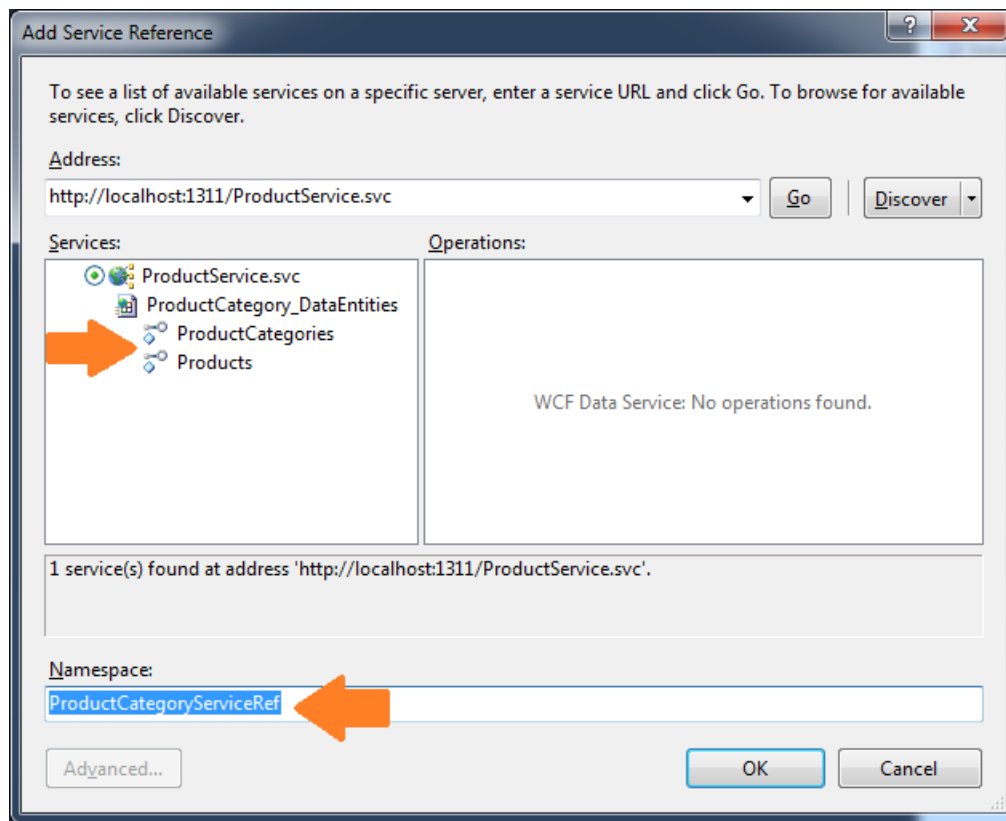
A WCF-es szolgáltatások elérésénél már szó esett arról, hogy a Visual Studio véletlenszerűen generált portokon keresztül hosztolja az ASP.NET alkalmazásunkat. Most megmutatom, hogyan lehet a véletlenszerű port helyett egy fix portot használni. Az ASP.NET projekt tulajdonságai között lehet ezt beállítani az alábbi módon: a Solution Explorerben jobb klikk a projekt nevére ➤ *Properties* ➤ *Web* fül a bal oldalon ➤ *Server szekció*. Itt válasszuk a *Specific port* lehetőséget, és hagyjuk meg az ott beírt port számot, ahogyan ezt a 8-19 ábra is mutatja!

A rendszer üzembe helyezésekor ez nem fog problémát okozni, ugyanis a Silverlight kliens relatív módon fog majd hivatkozni a szolgáltatásra.



8-19 ábra: Fix porton történő hosztolás beállítása

Vegyünk fel egy referenciát a Silverlight projektben a **ProductService** szolgáltatásra, és írjuk át a névteret **ProductCategoryServiceRef**-re! (8-20 ábra)



8-20 ábra: Szolgáltatás-referencia felvétele

A felhasználói felület most nagyon egyszerűen fog kinézni, egyetlen gombból és egy **DataGrid**-ből fog állni. A **LayoutRoot** kódja ezek alapján az alábbi módon kell, hogy kinézzen:

```
<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Button x:Name="btn_load_categories" Content="Kategóriák lekérése"
        Click="btn_load_categories_Click" />
    <sdk:DataGrid x:Name="dg_categories" Grid.Row="1" />
</Grid>
```

Nyissuk meg a **MainPage.xaml.cs** fájlt, és hivatkozzunk egy **using** direktívával a **WCFDataServices_Demo.ProductCategoryServiceRef** névtérre! Helyezzünk el egy privát adattagot **context** néven az osztályban, melynek a típusa legyen **ProductCategory_DataEntities**!

Ha megnézzük ennek az osztálynak a forráskódját (legyen kijelölve a típus, majd F12), akkor azt láthatjuk, hogy ez a **DataServiceContext** osztályból van származtatva, amely a **DataService** elérést megvalósító alap proxy osztály. Itt is aszinkron módon lehet lekérni az adatokat, viszont nem a WCF-nél látott formában, hanem a hagyományos .NET -es (**IASnyResult**) aszinkron minta mentén.

Ezek után inicializáljuk a **context** változót az alábbi módon:

```
context = new ProductCategory_DataEntities(new Uri("ProductService.svc", UriKind.Relative));
```

A gombra való kattintáskor először egy LINQ lekérdezést kell megfogalmaznunk, majd a **DataServiceQuery<T>** segédobjektum segítségével lekérdeznünk a szolgáltatástól az adatokat. A **T** itt egy entitást jelöl. Ennek a kódja így néz ki:

```
private void btn_load_categories_Click(object sender, RoutedEventArgs e)
{
    var query = from c in context.ProductCategories
                select c;
    DataServiceQuery<ProductCategory> dsq = (DataServiceQuery<ProductCategory>)query;
    dsq.BeginExecute(GetCategoriesCompleted, dsq);
}
```

Először megírjuk magát a lekérdezést, majd átalakítjuk **DataServiceQuery<ProductCategory>** típusúra (vagyis ez egy olyan DataServices-es lekérdezés, mely **ProductCategory** entitásokat használ), végül pedig meghívjuk a **BeginExecute** metódust, vagyis elindítjuk az aszinkron végrehajtást. A **BeginExecute** első paramétere egy aszinkron *callback* függvény, a második pedig egy tetszőleges objektum, amely jelen esetben maga a **dsq**, ugyanis ennek az **EndExecute** metódusán keresztül kapjuk majd meg a lekérdezés eredményét. Az aszinkron callback metódusnak az alábbi szignatúrával kell rendelkeznie:

```
void AsyncCallbackMethod(IAsyncResult iar)
```

Ezek alapján a **GetCategoriesCompleted** kódja az alábbi módon néz ki:

```
private void GetCategoriesCompleted(IAsyncResult iar)
{
    DataServiceQuery<ProductCategory> dsq =
        (DataServiceQuery<ProductCategory>)iar.AsyncState;
    IEnumerable<ProductCategory> categories = new
        ObservableCollection<ProductCategory>(dsq.EndExecute(iar));
    dg_categories.ItemsSource = categories;
}
```

Azért van szükségünk a **categories** köztes tárolóra, mivel a **DataGrid** nemcsak egyszer megy végig az elemeken, viszont az **EndExecute** metódus csak egyszer ad vissza **IEnumerable** értéket! Az alkalmazás futtatásakor a 8-21 ábrán látható felület fog fogadni bennünket.

Kategóriák lekérése					
ProductCategoryID	ParentProductCategoryID	Name	rowguid	ModifiedDate	Products
1		Bikes	cfbda25c-df71-47a7-b81b-64ee161aa37c	6/1/1998 12:00:00 AM	System.Data.Services.Client.Da
2		Components	c657828d-d808-4aba-91a3-af2ce02300e9	6/1/1998 12:00:00 AM	System.Data.Services.Client.Da
3		Clothing	10a7c342-ca82-48d4-8a38-46a2eb089b74	6/1/1998 12:00:00 AM	System.Data.Services.Client.Da
4		Accessories	2be3be36-d9a2-4eee-b593-ed895d97c2a6	6/1/1998 12:00:00 AM	System.Data.Services.Client.Da
5	1	Mountain Bikes	2d364ade-264a-433c-b092-4fcfb3804e01	6/1/1998 12:00:00 AM	System.Data.Services.Client.Da
6	1	Road Bikes	000310c0-bcc8-42c4-b0c3-45ae611af06b	6/1/1998 12:00:00 AM	System.Data.Services.Client.Da
7	1	Touring Bikes	02c5061d-ecdc-4274-b5f1-e91d76bc3f37	6/1/1998 12:00:00 AM	System.Data.Services.Client.Da

8-21 ábra: Termékkategóriák lekérése WCF Data Service segítségével

Valójában az eredménynek a 8-21 ábrán nem látható — a jobb oldalon lemaradó — része tartalmazza az érdekes információt, melyet a 8-22 ábra mutat be. Ha megnézzük ezt az ábrát, akkor azt láthatjuk, hogy **ProductCategory**-nak van egy **Products** tulajdonsága is (nagyon helyesen), amely egy üres gyűjteményre mutat (ez a lazy loading miatt üres), illetve van egy **ProductCategory1** és egy **ProductCategory2** tulajdonsága is. Utóbbi két tulajdonsággal azért rendelkezik ez az entitás, mivel az adatbázisban ez a tábla relációban áll önmagával (a **ParentProductCategoryID** mezőn keresztül) azért, hogy alkategóriákat is létre lehessen hozni. Ez az entitásmodellben pedig úgy reprezentálódik, hogy kapunk egy referenciát a reláció mindkét felére, ezért van a **ProductCategory** tulajdonságból kettő is.

Kategóriák lekérése							
Pri	Pare	Name	rowgu	Modified	Products	ProductCategory1	ProductCategory2
1		Bikes	cfbda2	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	
2		Compo	c65782	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	
3		Clothin	10a7c34	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	
4		Accesso	2be3	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	
5	1	Mounta	2d364a	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	
6	1	Road Bi	000310	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	
7	1	Touring	02c	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	
8	2	Handle	3ef2c72	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	
9	2	Bottom	a9e540	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	
10	2	Brakes	d43had	6/1/1998	System.Data.Services.Client.DataServiceCollection`1[WCFDataServices_Demo.ProductCategoryServiceRef.Product]	System.Data.Services	

8-22 ábra: A Products, ProductCategory1 és ProductCategory2 tulajdonságok

Master-Details nézet létrehozás a WCF Data Services segítségével

Alakítsuk át úgy az előző példaalkalmazást, hogy a kapcsolódó termékeket meg is lehessen tekinteni a kiválasztott kategóriánál! Ehhez némileg át kell írunk a **DataGrid** kódját, melynek részletezésébe most nem mennék bele (a 11. fejezet foglalkozik majd ezzel). Most bőven elég annyit tudnunk róla, hogy a gombra való kattintáskor megkapjuk a kiválasztott sor azonosítóját. Íme, a **DataGrid** új kódja:

```
<sdk:DataGrid x:Name="dg_categories" Grid.Row="1" AutoGenerateColumns="False">
  <sdk:DataGrid.Columns>
    <sdk:DataGridTextColumn Binding="{Binding Name}" Header="Kategória neve" />
    <sdk:DataGridTextColumn Binding="{Binding ProductCategoryID}"
      Header="Kategória azon." />
    <sdk:DataGridTemplateColumn Header="Kategória termékei">
      <sdk:DataGridTemplateColumn.CellTemplate>
        <DataTemplate>
          <Button x:Name="btn_details" Tag="{Binding ProductCategoryID}"
            Content="Kategória termékei" Click="btn_details_Click" />
        </DataTemplate>
      </sdk:DataGridTemplateColumn.CellTemplate>
    </sdk:DataGridTemplateColumn>
  </sdk:DataGrid.Columns>
</sdk:DataGrid>
```

A **Kategória termékei** gomb **Click** eseménykezelőjében először ki kell nyernünk a kiválasztott sor elsődleges kulcsát, majd ezt át kell adnunk a **DetailsChildWindow** konstruktorának, végül pedig meg kell hívunk a gyerekablak **Show** metódusát. (A **DetailsChildWindow**-t pillanatokon belül elkészítjük). Íme, ennek a kódja:

```
private void btn_details_Click(object sender, RoutedEventArgs e)
{
    int cid = Convert.ToInt32(((Button)sender).Tag);
    DetailsChildWindow dcw = new DetailsChildWindow(cid);
    dcw.Show();
}
```

Adjunk a projektünkhöz egy új **Silverlight Child Window** vezérlőt **DetailsChildWindow** néven! A hozzá tartozó cs fájlban hozzunk létre egy új privát adattagot, amiben majd eltárolhatjuk a konstruktorban paraméterként kapott kategóriaazonosítót! Ennek megfelelően írjuk át a konstruktort, így eme lépésnek kódja az alábbi:

```
private int cid;
public DetailsChildWindow(int cid)
{
    InitializeComponent();
    this.cid = cid;
}
```

A felhasználói felületen a **LayoutRoot** első sorába helyezzük el az alábbi kódot (a MainPage oldalról másoljuk át az sdk névtérhez szükséges **xmlns** bejegyzést is):

```
<sdk:DataGrid x:Name="dg_products" AutoGenerateColumns="False">
  <sdk:DataGrid.Columns>
    <sdk:DataGridTextColumn Binding="{Binding ProductID}" Header="Termék azon." />
    <sdk:DataGridTextColumn Binding="{Binding Name}" Header="Termék neve" />
    <sdk:DataGridTextColumn Binding="{Binding Color}" Header="Szín" />
    <sdk:DataGridTextColumn Binding="{Binding Size}" Header="Méret" />
    <sdk:DataGridTextColumn Binding="{Binding Weight}" Header="Súly" />
  </sdk:DataGrid.Columns>
</sdk:DataGrid>
```

Ezek után térjünk vissza a **.cs** fájlhoz, és hozzunk létre két újabb privát adattagot, egyet a kontextusnak, egyet pedig a termékeknek! Utóbbihoz egy speciális tárolót, a **DataServiceCollection<T>** generikus osztályt használjuk. Íme, a két adattag:

```
private ProductCategory_DataEntities context;
private DataServiceCollection<Product> products;
```

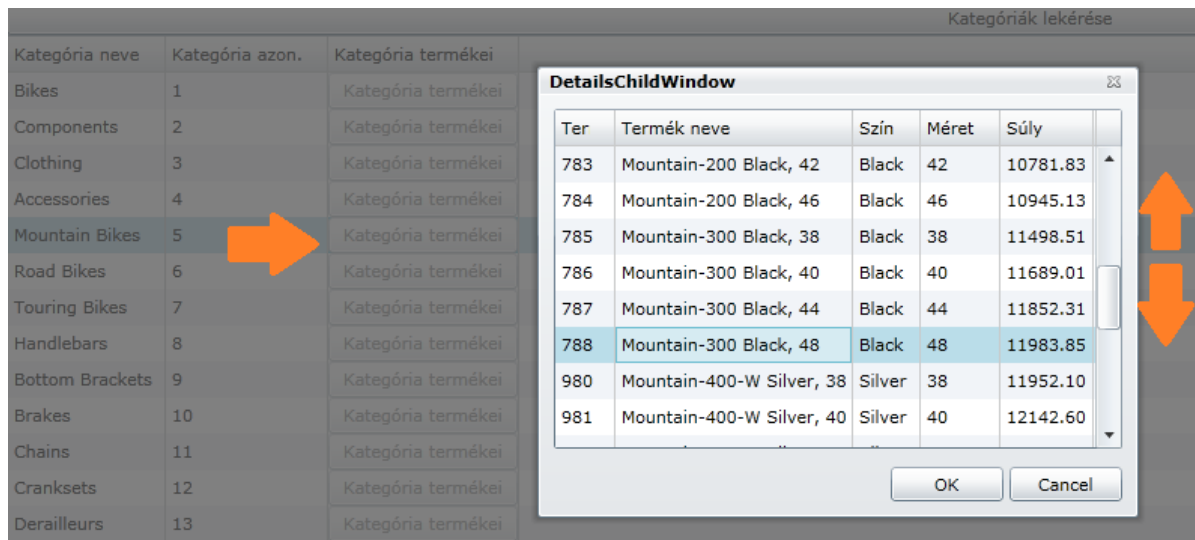
Inicializáljuk őket a konstruktorban, majd hívjuk a mindjárt megírásra kerülő **LoadProducts** metódust:

```
...
this.cid = cid;
context = new ProductCategory_DataEntities(new Uri("ProductService.svc",
    UriKind.Relative));
products = new DataServiceCollection<Product>();
LoadProducts();
```

A **LoadProducts** elején LINQ segítségével fogalmazzuk meg a lekérdezést a megfelelő termékekre, majd pedig a **DataServiceCollection Load** metódusát hívjuk meg aszinkron módon a lekérdezéssel felparaméterezve. Ez a WCF-nél megszokott **LoadCompleted** eseményből és **LoadAsync** metódus párosból áll. Íme, ennek a kódja:

```
private void LoadProducts()
{
    var query = from p in context.Products
                where p.ProductCategoryID == cid
                select p;
    products.LoadCompleted += (s, e) => {
        if (e.Error == null)
            dg_products.ItemsSource = products; };
    products.LoadAsync(query);
}
```

Itt az eredményt nem az **e.Result**-ban fogjuk megkapni, hanem a **products**-ban, hiszen azon hívtuk a feltöltő (Load) metódust. Ezzel el is készültünk az alkalmazással. Futtassuk és kattintsunk rá, mondjuk az ötös azonosítóval rendelkező sor gombjára! Az eredmény a 8-23 ábrán látható.



8-23 ábra: Master-Details nézet

Fontos: Megjegyezném, hogy ez a példaalkalmazás rendkívüli módon pazarolja a hálózati erőforrásokat, ezért célszerű itt is bevezetni a DTO-kat. Viszont a dolog érdekessége, hogy WCF Data Services-nél a kliensoldalon kell definiálni az objektumokat (hiszen itt vannak a LINQ lekérdezések), és nincs szükség a **DataContract**, illetve **DataMember** attribútumokra sem.

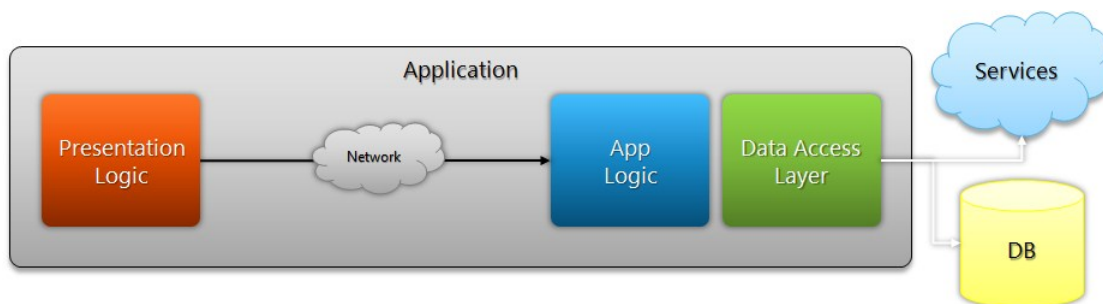
WCF RIA Services, a kliens-szerver modell újragondolása

Az előző két alfejezetben két teljesen különböző elgondolást —megközelítést— láthattunk. Ebben az alfejezetben egy olyan rendszerrel fogunk megismerkedni, mely mindkét technológiából csipeget egy kicsit, és kiegészíti sok-sok egyéb hasznos szolgáltatással.

A Nagy Ötlet, avagy a magától adódó architektúra?

Amikor kliens-szerver modellben gondolkodunk, akkor általában bevezetünk egy köztes réteget is (szolgáltatás réteg), amelyen keresztül a két komponens kommunikálhat egymással. Ez a réteg azért felelős, hogy kiajánlja a megfelelő metódusokat a kliensek felé, illetve fogadja a tőlük érkező kéréseket. Mi lenne, ha ezt a réteget nem nekünk kellene megírni, hanem a rendszer gondoskodna a kliens-szerver közötti kommunikációról, és nekünk csak az üzleti logikával és az adateléréssel kellene foglalkoznunk?

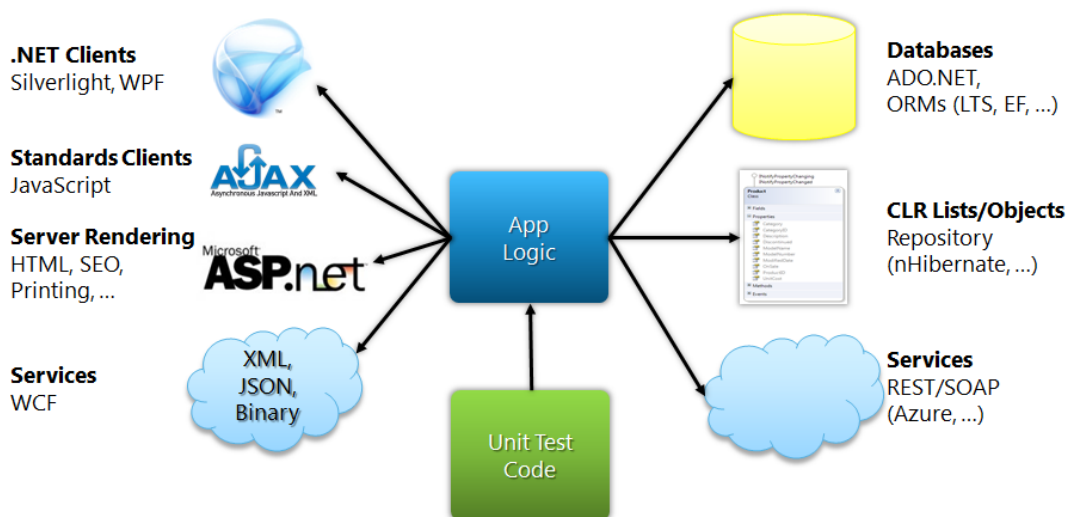
A **WCF RIA Services** ezt a problémát veszi le a vállunkról oly módon, hogy a kliensoldalra mint a szerveroldal egy kiegészítésére tekint. Másképpen megfogalmazva, a kliensoldal az alkalmazás reprezentációs rétegeként funkcionál. Ezt a szemléletet mutatja be a 8-24 ábra.



8-24 ábra: A WCF RIA Services filozófiája

A **WCF RIA Services** tehát teljes körű kiszolgálást nyújt a szolgáltatás réteg hatékony kezelésében, kezdve a szolgáltatás felületének definiálástól, a DTO-k generálásán keresztül, egészen a szolgáltatás elérésig.

A WCF RIA Services emellett end-to-end támogatást nyújt olyan gyakori feladatokhoz, mint például az adatok érvényességének vizsgálata, a felhasználó-azonosítás, jogosultságkezelés, stb. Ehhez egy rendkívül erős integrációra van szükség a szerver és a kliens oldal között. Ez ASP.NET és Silverlight esetén viszonylag egyszerűen megoldható, de a WCF RIA Services ennél sokkal általánosabb. A megjelenítési réteg sokféle lehet, mint ahogyan azt a 8-25 ábra is szemlélteti.

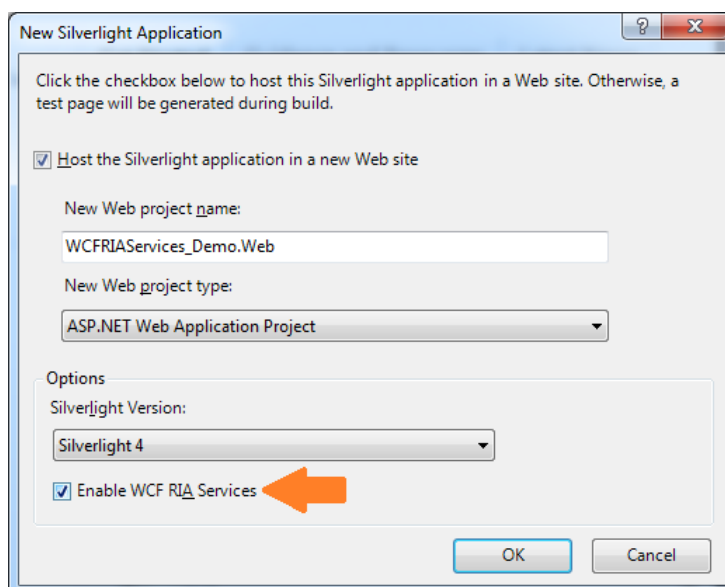


8-25 ábra: A WCF RIA Services felépítése

Összegezve tehát, a WCF RIA Services leegyszerűsíti a webes többretegű alkalmazások fejlesztését.

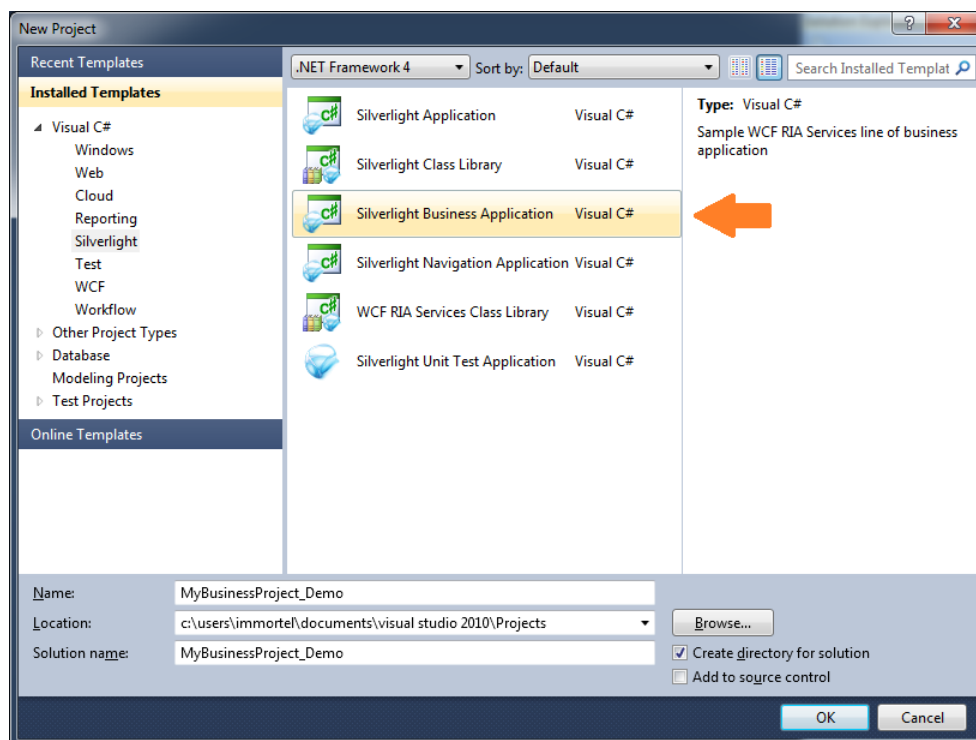
Szolgáltatások létrehozása

Az egyik alapvető különbség az eddig látott *szolgáltatások létrehozása* részekhez képest az, hogy itt már a projekt létrehozásakor meg kell adnunk, hogy ez egy WCF RIA Services-t használó projekt lesz. Ezzel létrejön az úgynevezett *RIA link*, ami egy kapocs a Silverlight kliens és az ASP.NET szerver projekt között. Kétféleképpen hozhatunk létre ilyen projektet. Az egyik lehetőség, hogy a Silverlight Application típusú projektsablont választjuk, és a hoszt projekt megadásakor bepipáljuk az *Enable WCF RIA Services* opciót, amint azt a 8-26 ábra mutatja.



8-26 ábra: RIA link engedélyezése

A másik lehetőség, hogy a *Silverlight Business Application* típusú projektsablont választjuk (8-27 ábra). A kettő között az a nagy különbség, hogy a Business Application változatnál nagyon sok előre megírt funkciót kapunk alpból a projekthez, például felhasználó-azonosítást, lokalizáció kezelést és még sok egyéb dolgot. Mivel ezekre ebben a példában nem lesz szükségünk, ezért inkább maradunk az első változatnál.



8-27 ábra: A *Silverlight Business Application* projektsablon kiválasztása

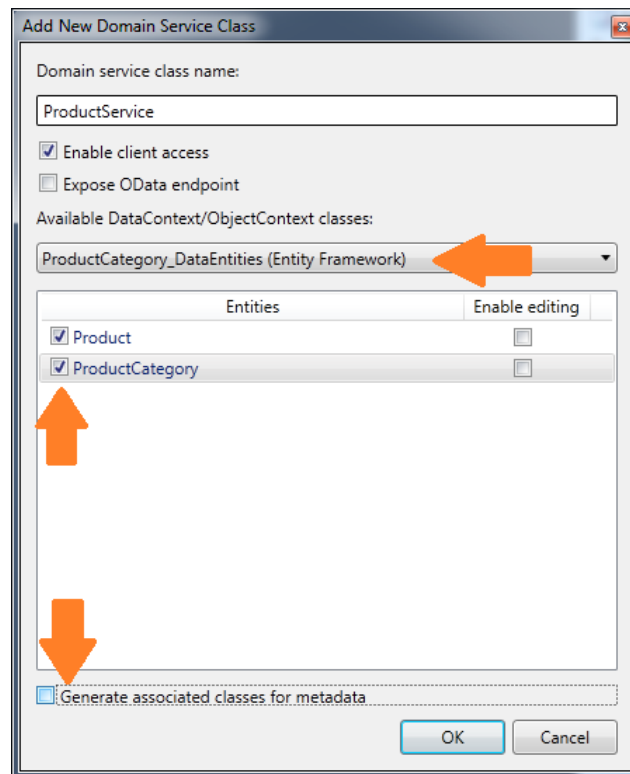
Tehát hozzunk létre egy új Silverlight Application típusú alkalmazást, ahol engedélyezzük a RIA linket, és adjuk neki a **WCFRIAServices_Demo** nevet!

A következő eltérés az eddig látottakhoz képest, hogy itt nem egy, a technológiával megegyező nevű szolgáltatássablont kell hozzáadnunk a projektünkhöz, hanem egy ún. **DomainService**-t. A **DomainService** az egyszerű WCF szolgáltatás és a WCF Data Services keveréke. Ez az osztály a **LinqToEntitesDomainService<T>** generikus osztályból származik (Data Services vonal), viszont metódusokat kell benne definiálni (mint az egyszerű WCF szolgáltatás). Az őosztály nevéből látszik, hogy ez is inkább az Entity Frameworköt preferálja, nem pedig a LINQ-et.

A **DomainService** létrehozásakor — a Data Services-zel ellentétben — van lehetőségünk LINQ adatmodell használatára is, ehhez viszont le kell tölteni a *WCF RIA Services Toolkit*-et az alábbi címről:
<http://www.microsoft.com/downloads/en/details.aspx?FamilyID=c4f02797-5f9e-4acf-a7dc-c5ded53960a6&displaylang=en>.

Jó szokásunkhoz híven ismét az AdventureWorks Lite adatbázissal fogunk dolgozni, és a benne lévő **Product**, illetve **ProductCategory** táblákat fogjuk használni a demóban. Hozzuk létre a **ProductCategory.edmx** fájlt a WCF Data Services-es *Szolgáltatás létrehozása* részben bemutatott módon, majd fordítsuk le a projektet!

Ezek után adjunk egy új *Domain Service Class* típusú fájlt a projekthez **ProductService.cs** néven! A művelet végrehajtása során megjelenő dialógusban az *Available DataContext/ObjectContext classes* résznél ha mindent jól csináltunk a legördülő listában a **ProductCategory_DataEntities** bejegyzésnek kell kiválasztva lennie. Az alatta lévő táblázatban jelöljük ki az összes entitást! Utolsó teendőként pedig a legalsó sorban lévő *Generate associated classes for metadata* szolgáltatást kapcsoljuk ki (8-28 ábra).



8-28 ábra: Az új DomainService beállításai

A generált fájlban két metódusunk lesz, a **GetProducts** és a **GetProductCategories**, melyek mindegyike egy **IQueryable<T>** példánnyal tér vissza. Ezeket a műveket *query method*-oknak hívják a WCF RIA Services terminológiában. Bármilyen prefixszel kezdődhetnek és tetszőleges számú paraméterük is lehet, viszont a visszatérési értékük csak **IQueryable<T>** vagy **IEnumerable<T>** lehet. Ezeken belül tetszőleges *Linq2Entities* lekérdezéseket írhatunk, ahol a **DataContext**-et az osztály **ObjectContext** tulajdonsága szolgáltatja. (Lehetőség van egyéb metódusok definiálására is, de azokról majd később).

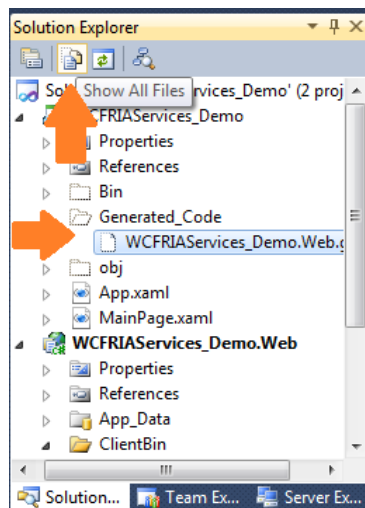
Készítsünk el egy saját metódust, amelyen keresztül le lehet kérdezni a termékeket kategória szerint. Ennek a kódja az alábbi módon néz ki:

```
public IQueryable<Product> GetProductsByCategory(int cid)
{
    return this.ObjectContext.Products.Where(p => p.ProductCategoryID == cid);
}
```

Ezzel el is készültünk a szolgáltatásunkkal.

Szolgáltatások elérése és használata

A **ProductService** szolgáltatás elérése nagyon egyszerű feladat, mindösszesen csak le kell fordítani a Solutiont, és a szolgáltatás máris elérhetővé válik a kliensoldalon. Ha ezek után a Solution Explorerben a *Show All Files* ikonra kattintunk, akkor láthatjuk, hogy kaptunk egy **Generated_Code** mappát, illetve benne egy **WCFRIAService_Demo.Web.g.cs** fájlt, ahogyan azt a 8-29 ábra is mutatja.



8-29 ábra: Generate_Code rejtett mappa és a tartalma

Nyissuk meg ezt a fájlt, és tekintsük meg a benne lévő négy osztálydefiníciót! Találni fogunk itt egy **Product** és egy **ProductCategory** osztályt, melyek az entitásaikat reprezentálják, illetve lesz egy **WebContext** és egy **ProductContext** osztály is. A **WebContext** a felhasználók azonosítására használható, míg a **ProductContext** a **DomainService**-hez tartozó proxy osztály. Ha utóbbit közelebbről is megvizsgáljuk, akkor azt látjuk, hogy ami a szerveroldalon query method volt, az itt egy olyan függvény lett, aminek a visszatérési értéke **EntityQuery<T>**, és a neve végére oda került a **Query** tag. Egy másik fontos dolog ezzel az osztállyal kapcsolatban az **EntityContainer**, amely az adatbázisunkat reprezentálja, illetve a benne lévő adattáblákat **EntitySet**-ekként.

A **ProductContext** osztály kétféleképpen is használható, az egyik esetben imperatív kódot (C#), a másikban pedig deklaratív kódot (XAML) kell írunk. Nézzük először az imperatív megközelítést!

Egy olyan demóalkalmazást fogunk most elkészíteni, amely egy legördülő listába feltölti a termékkategóriákat, majd a kiválasztott elemhez tartozó termékeket betölti egy **DataGrid**-be. A kódolás előtt készítsük el a felhasználói felületet, melynek a kódja az alábbi módon néz ki:

```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <ComboBox x:Name="cb_categories" Width="100"
    DisplayMemberPath="Name" SelectedValuePath="ProductCategoryID"/>
  <sdk:DataGrid x:Name="dg_products" Grid.Row="1" />
</Grid>
```

Nyissuk meg a **MainPage.xaml.cs** fájlt, és iratkozzunk fel a konstruktorban a **Loaded** eseményre:

```
public MainPage()
{
    InitializeComponent();
    Loaded += MainPage_Loaded;
}
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    throw new NotImplementedException();
}
```

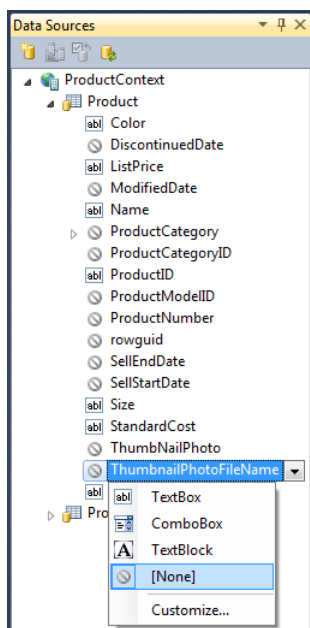
Adjunk az osztályhoz egy új privát adattagot **context** néven, melynek a típusa legyen **ProductContext**! A **MainPage_Loaded** eseménykezelőben szükségünk lesz majd egy **LoadOperation<T>** segédosztályra is,

ezért a **using** direktíva segítségével emeljük be a **System.ServiceModel.DomainServices.Client** névteret! A **Loaded** metódus kódja ezek után legyen az alábbi:

```
context = new ProductContext();
LoadOperation<ProductCategory> lo = context.Load(context.GetProductCategoriesQuery());
cb_categories.ItemsSource = lo.Entities;
```

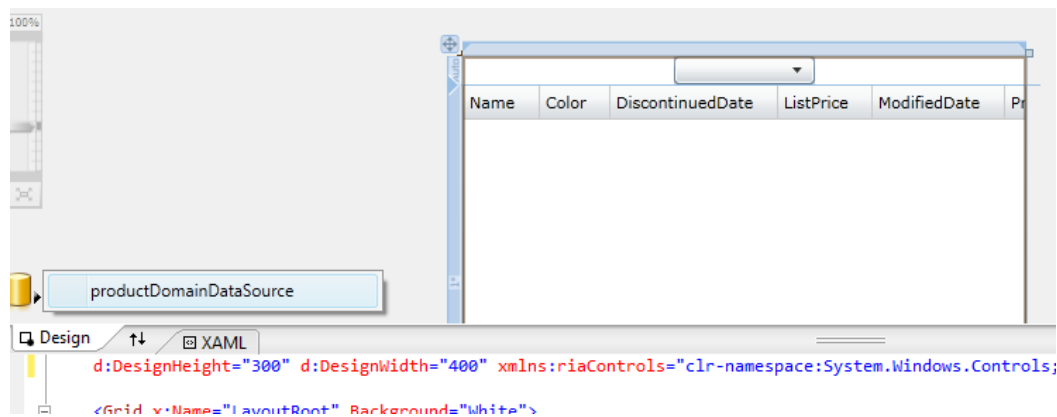
Itt először inicializáljuk a **context**-et, utána pedig meghívjuk a **GetProductCategories** metódust, melynek az eredményét egy **LoadOperation** példány **Entities** tulajdonságán keresztül érhetjük el.

Ezzel el is készültünk a kódolás résszel. Nézzük meg most a grafikus felület segítségével történő XAML kód generálást! A részleteket megjelenítő nézetet egy úgynevezett **DomainDataSource** vezérlő segítségével szeretnénk elkészíteni. Ezen keresztül deklaratív módon tudunk meghívni metódusokat, átadni paramétereket, sőt, még szűrni, csoportosítani, rendezni és lapozni is. Ehhez a **DataSources** ablakra lesz szükségünk, melyet a **View** menüből érhetünk el. Első kinyitáskor kicsit lassú lesz, ugyanis létre kell hoznia egy **WCFRIAServices_Demo.Web.ProductContext.datasource** nevű fájlt a **Properties/DataSources** mappán belül. Miután végzett a generálással, nyissuk ki a **Product** node-ot és a **ProductId**, **Name**, **StandardCost**, **ListPrice**, **Color**, **Weight**, **Size** tulajdonságok kivételével mindegyik reprezentációját állítsuk **None**-ra. Ezt úgy tudjuk megtenni, hogy rákattintunk az adott mezőre, és ott a legördülő listában a **None** lehetőséget választjuk ki, ahogyan azt a 8-30 ábra is mutatja.



8-30 ábra: A megjelenítendő tulajdonságok összeválogatása

Ezek után fogjuk meg a **Product** csomópontot, és húzzuk a tervezőfelületre! (A korábban kézzel begépett **DataGrid**-et töröljük ki előtte!) A 8-31 ábra mutatja a tervezési időben a felületet.



8-31 ábra: A tervezőfelület

A **DomainDataSource** kódját változtassuk meg egy kicsit! Nem egyszerűen az összes terméket akarjuk lekérni, hanem a kiválasztott kategória alapján szeretnénk szűrni őket. Vagyis, a **QueryName** tulajdonságát kell átírnunk **GetProductsByCategoryQuery**-ra. Bár IntelliSense támogatást itt nem kapunk, de ha a **ProductContext** egy nem létező függvényét akarnánk meghívni, akkor a XAML editor aláhúzással jelezne nekünk a helytelen metódusnevet.

Az újonnan beállított függvénynek van egy paramétere, melynek aktuális értékét a legördülő listából szeretnénk kinyerni, ezért **Element2Element Binding**-ot fogunk hozzá használni. Íme, a paraméter megadásának kódja:

```
<riaControls:DomainDataSource.QueryParameters>
  <riaControls:Parameter ParameterName="cid"
    Value="{Binding ElementName=cb_categories, Path=SelectedValue}" />
</riaControls:DomainDataSource.QueryParameters>
```

Mivel alaphoz az adatkötés **OneWay** típusú, ezért amint megváltozik a kiválasztott elem a legördülő listában, azonnal újra meg fogja hívni a WCF RIA Services a **GetProductsByCategory** metódust.

Ezzel el is készültünk a demóval. Íme, összefoglalásképpen a **LayoutRoot** teljes kódja:

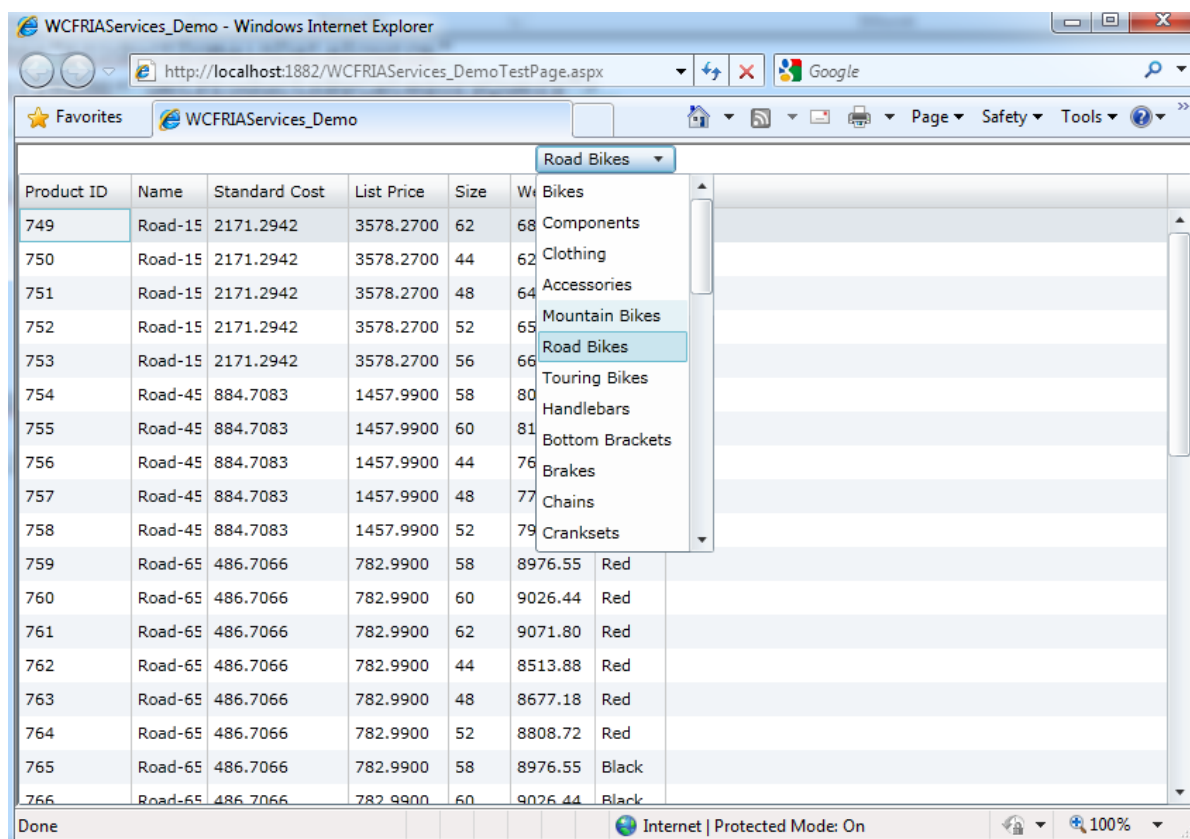
```
<Grid x:Name="LayoutRoot" Background="White">
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
  <ComboBox x:Name="cb_categories" Width="100" DisplayMemberPath="Name"
    SelectedValuePath="ProductCategoryID"/>
  <sdk:DataGrid x:Name="productDataGrid" AutoGenerateColumns="False" Grid.Row="1"
    ItemsSource="{Binding ElementName=productDDS, Path=Data}">
    <sdk:DataGrid.Columns>
      <sdk:DataGridTextColumn x:Name="productIDColumn"
        Binding="{Binding Path=ProductID, Mode=OneWay}"
        Header="Product ID" IsReadOnly="True"/>
      <sdk:DataGridTextColumn x:Name="nameColumn"
        Binding="{Binding Path=Name}" Header="Name"/>
      <sdk:DataGridTextColumn x:Name="standardCostColumn"
        Binding="{Binding Path=StandardCost}" Header="Standard Cost"/>
      <sdk:DataGridTextColumn x:Name="listPriceColumn"
        Binding="{Binding Path=ListPrice}" Header="List Price"/>
      <sdk:DataGridTextColumn x:Name="sizeColumn"
        Binding="{Binding Path=Size}" Header="Size"/>
      <sdk:DataGridTextColumn x:Name="weightColumn"
        Binding="{Binding Path=Weight}" Header="Weight" />
      <sdk:DataGridTextColumn x:Name="colorColumn"
        Binding="{Binding Path=Color}" Header="Color"/>
    </sdk:DataGrid.Columns>
  </Grid>
```

```

</sdk:DataGrid>
<riaControls:DomainDataSource AutoLoad="True"
  d:DesignData="{d:DesignInstance my:Product, CreateList=true}" Height="0" Width="0"
  LoadedData="productDomainDataSource_LoadedData" Name="productDDS"
  QueryName="GetProductsByCategoryQuery">
  <riaControls:DomainDataSource.QueryParameters>
    <riaControls:Parameter ParameterName="cid"
      Value="{Binding ElementName=cb_categories, Path=SelectedValue}" />
  </riaControls:DomainDataSource.QueryParameters>
  <riaControls:DomainDataSource.DomainContext>
    <my:ProductContext />
  </riaControls:DomainDataSource.DomainContext>
</riaControls:DomainDataSource>
</Grid>

```

A 8-32 ábra egy futás közben elcsípett képernyőképet mutat.

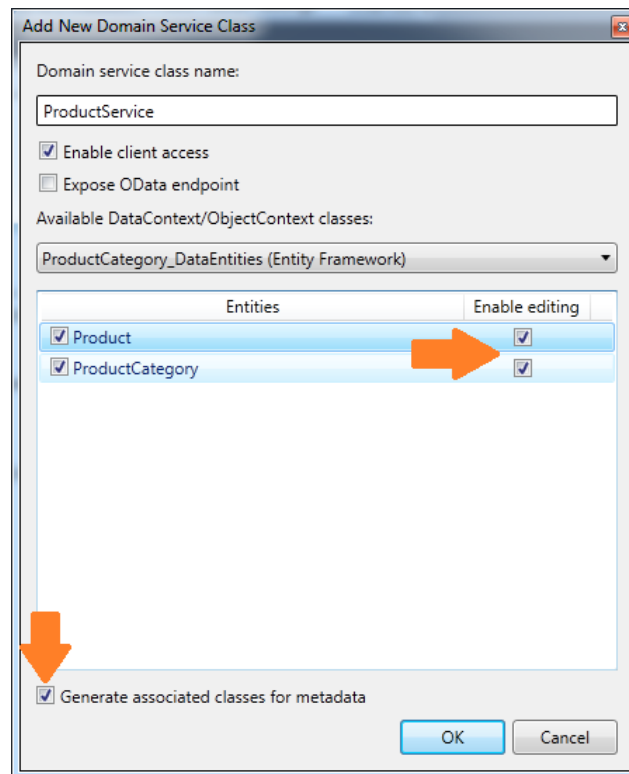


8-32 ábra: Egy master-details nézet a WCF RIA Services segítségével

Adatok módosítása és validálása WCF RIA Service-n keresztül

Az eddig elkészített demó alkalmazások mindegyike csak az adatok megjelenítésével foglalkozott. Most nézzünk meg egy olyat, amely lehetőséget nyújt a felhasználónak az adatok szerkesztésére, és ellenőrzi is az általa megadott új értékek helyességét.

Ehhez az előző részben elkészített demóalkalmazást fogjuk átalakítani. Mivel a programozó alapjában véve lusta, ezért ha nem kell, akkor nem kódol feleslegesen. Így van ez a WCF RIA Services esetén is, ahol bár mi magunk megírhatnánk a CRUD műveletekhez tartozó kódokat, de ha azt le is generáltathatjuk, akkor miért ne tennénk? Töröljük ki a **ProductService.cs** fájlt, és adjuk hozzá újból a projekthez, de most úgy, hogy az entitások szerkeszthetők legyenek — mindegyik entitásnál pipáljuk be az *Enable editing* opciót is — ,és generáltassunk hozzájuk egy segédosztályt is, amelyben az entitásokat leíró metaadatokat tárolhatjuk — a *Generate associated classes for metadata* opció kijelölésével! (8-33 ábra)



8-33 ábra: Szerkeszthető entitások létrehozása

A **ProductService** osztályunk most már 8 metódussal büszkélkedhet. Amit ezekről érdemes megjegyezni, hogy a frissítés, törlés és beszúrás függvények prefixe és paraméterlistája kötött (lásd az alábbi weblapon: [http://msdn.microsoft.com/en-us/library/ee707373\(VS.91\).aspx](http://msdn.microsoft.com/en-us/library/ee707373(VS.91).aspx)). Létezik ún. nevesített frissítés (*named update*) is, melynek lényege, hogy az entitás paraméteren kívül lehet tetszőleges számú egyéb paraméter is. Természetesen írhatunk olyan metódusokat is, melyek nem entitásokon végeznek műveleteket, viszont ezeket az ún. **Invoke** attribútummal el kell látni.

Másoljuk be ismét a szolgáltatásban a **GetProductsByCategory** függvényt, hogy a kliens oldalról továbbra is el lehessen érni ezt a funkciót! Ezek után nyissuk meg a **ProductService.metadata.cs** fájlt! Ebben a **Product** és a **ProductCategory** osztályok vannak kibővítvé két új **internal** osztállyal (**ProductMetadata**, **ProductCategoryMetadata**). Utóbbiak arra szolgálnak, hogy az egyes tulajdonságokat elláthassuk érvényességi feltételeket leíró attribútumokkal. Vagyis deklaratív módon tehetünk megkötéseket az egyes mezőkre. Ezeket az attribútumokat gyűjtőnéven *adat annotációknak* (*data annotations*) hívják.

Az egyes tulajdonságokra többek között olyan megszorításokat tehetünk, mint például annak kötelező kitöltése (**Required**), a szöveg maximális hosszúsága (**StringLength**) vagy akár a mező tartalmának típusa (**DataType**). (Ezen a weblapon található az annotációk teljes listája: [http://msdn.microsoft.com/en-us/library/dd901590\(VS.95\).aspx](http://msdn.microsoft.com/en-us/library/dd901590(VS.95).aspx)).

Helyezzünk el néhány megkötést a **Product** osztályon!

```
internal sealed class ProductMetadata
{
    private ProductMetadata() { }

    [DataType(DataType.Currency)]
    public decimal ListPrice { get; set; }

    [Required]
    [StringLength(50, MinimumLength=1)]
    public string Name { get; set; }
    [ReadOnly(true)]
    public int ProductID { get; set; }
```

```
[Range(typeof(DateTime), "2000.01.01", "2011.01.01")
public DateTime SellStartDate { get; set; }
..
}
```

Fordítsuk le az alkalmazást, hogy a kliensoldalon frissüljön a proxy osztály, és ezzel el is készültünk a szerveroldallal. A kliens oldalon két dolgunk van még hátra. Az egyik, hogy a DataGrid mezőlistájába felvegyük a **SellStartDate** mezőt, ennek a kódja az alábbi:

```
<sdk:DataGridTextColumn x:Name="sellStartDateColumn" Binding="{Binding Path=SellStartDate}"
Header="SellStartDate" />
```

A másik pedig, hogy feliratkozunk a DataGrid **RowEditEnded** eseményére és kezeljük azt. A feliratkozáshoz tartozó XAML kód:

```
<sdk:DataGrid Name="productDataGrid" ... RowEditEnded="productDataGrid_RowEditEnded">
```

Az eseményt kezelő függvény kódja:

```
private void productDataGrid_RowEditEnded(object sender, DataGridRowEditEndedEventArgs e)
{
    if (!productDDS.IsSubmittingChanges)
    {
        if (productDDS.HasChanges)
            productDDS.SubmitChanges();
    }
}
```

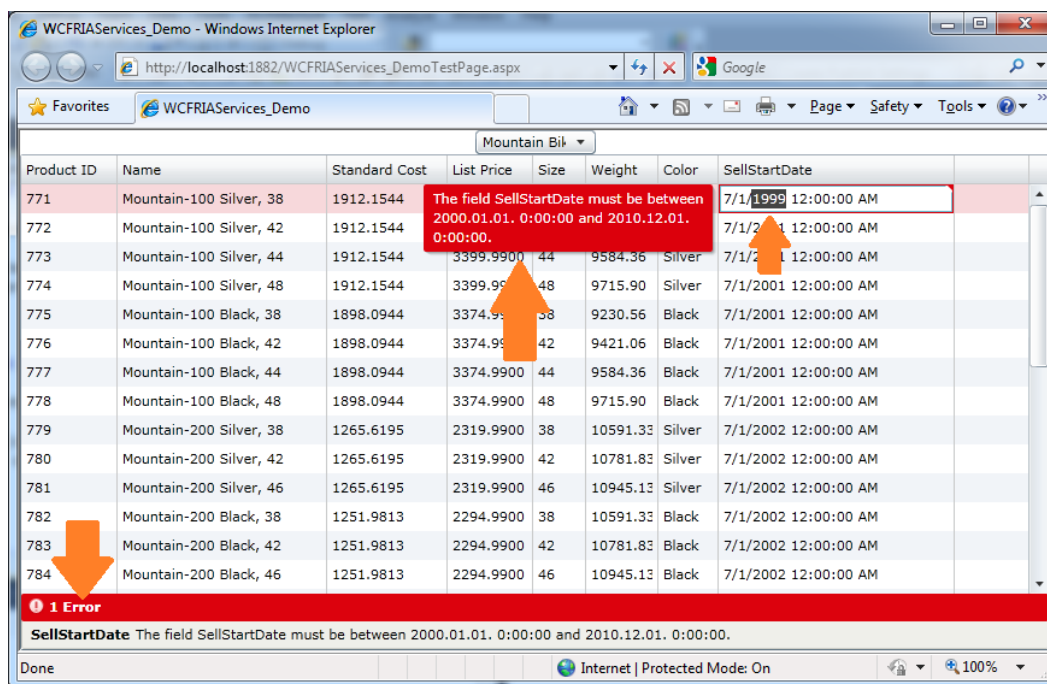
A függvény törzsében először lekérdezzük, hogy nem éppen egy **Submit** művelet közben próbáljuk-e megzavarni a **DomainDataSource**-t. Ha nem, akkor megnézzük azt, hogy történt-e egyáltalán változás, amit el kellene menteni. Ha van változás, akkor azt felküldjük az adatbázisba.

Amennyiben szeretnénk értesítést kapni a mentés sikerességének kimeneteléről, abban az esetben fel kell iratkoznunk a **DataSource** objektum **SubmittedChanges** eseményére. Ennek az **EventArgs** objektumán keresztül megtudhatjuk, hogy történt-e hiba a változásokat visszaíró művelet végrehajtása során (**HasError** tulajdonság). Sőt, még azt is lekérdezhetjük, hogy melyik entitásokkal volt probléma (**EntitiesInError** tulajdonság). Ennek persze csak akkor van értelme, ha a *unit-of-work* mintát használva nem egyesével mentjük el a módosított elemeket az adatbázisba, hanem egyszerre többet küldünk fel. Az ehhez tartozó eseménykezelő kódja az alábbi módon nézne ki:

```
private void productDSS_SubmittedChanges(object sender, SubmittedChangesEventArgs e)
{
    if (e.HasError)
    {
        foreach (var item in e.EntitiesInError)
        {
            // hibak feldolgozasa
        }
        MessageBox.Show(e.Error.Message);
    }
}
```

Ezzel el is készültünk az alkalmazásunkkal. Próbáljuk is ki: kérdezzünk le egy olyan kategóriát, amelyhez tartoznak termékek, és próbáljunk meg egy olyan mezőt módosítani, amelyre tettünk megszorítást! Olyan

értéket állítsunk be neki, amely a megszorítást megsérti! Például a Mountain Bike kategóriába tartozó valamely rekord **SellStartDate** mezőjében írjuk át az évet 1999-re! Az eredményt a 8-34 ábra mutatja.



8-34 ábra: Validáció hiba

Mivel DataGrid-et használunk, ezért az automatikusan kiértékeli a metaadatoknál megadott ellenőrzési feltételeket, és ha valami nem stimmel, akkor azt egyből jelzi a felhasználó felé.

Fontos: Ennek a fejezetnek nem célja a technológiák részletes ismertetése, ezért ajánlott az alábbi oldalak felkeresése további információkért:

- WCF: [http://msdn.microsoft.com/en-us/library/cc296254\(v=vs.95\).aspx](http://msdn.microsoft.com/en-us/library/cc296254(v=vs.95).aspx)
- WCF Data Services: <http://msdn.microsoft.com/en-us/library/cc668792.aspx>
- WCF RIA Services: [http://msdn.microsoft.com/en-us/library/ee707344\(v=vs.91\).aspx](http://msdn.microsoft.com/en-us/library/ee707344(v=vs.91).aspx).

További kommunikációs lehetőségek röviden

Ebben az alfejezetben néhány kódrészlet felvillantásával szeretném bemutatni az egyes technikák alapjait, emiatt itt nem fogunk teljes példaalkalmazásokat elkészíteni. Céлом ezzel az, hogy segítsék a lehetőségek megismerésében.

HTTP alapú kommunikáció a WebClient osztály segítségével

Az eddig látott demókban erőforrás alatt mindig adatbázist értettünk. Most azt az esetet vizsgáljuk meg, amikor az erőforrás egy szövegfájl, esetleg egy kép vagy akár egy tömörített állomány. Ezek le- és feltöltésének két módja létezik Silverlightban. Az egyik lehetőség az alacsony szintű **GET/POST** utasítások kezelése a **HttpRequest** és **HttpResponse** objektumok segítségével. A másik lehetőség pedig az ezek fölé írt csomagoló osztály, a **WebClient** használata. Ebben a részben mi csak az utóbbi esettel fogunk foglalkozni.

A **WebClient** osztály kétféle erőforrást különböztet meg: a szöveget, illetve a nem szöveg alapút. Nézzük meg először a szöveges erőforrások kezelését! A le-, illetve feltöltésükhöz egy-egy metódus és egy-egy esemény tartozik, amelyekből sejthető, hogy a kommunikáció aszinkron módon történik a szerver és a kliens között. Letöltés esetén a **DownloadStringAsync** metódust kell meghívni egy **Uri** példánnyal felparaméterezve, amely tartalmazza a lekérendő erőforrás egyedi azonosítóját. A művelet meghívása

előtt fontos feliratkozni a hozzá tartozó **DownloadStringCompleted** eseményre. Íme, egy egyszerű példa a használatára:

```
private void btn_get_files_Click(object sender, RoutedEventArgs e)
{
    WebClient client = new WebClient();
    client.DownloadStringCompleted += client_DownloadStringCompleted;
    client.DownloadStringAsync(new Uri("filelist.xml", UriKind.Relative));
}
private void client_DownloadStringCompleted(object sender,
    DownloadStringCompletedEventArgs e)
{
    if (e.Error == null)
    {
        XDocument doc = XDocument.Parse(e.Result);
        var result = from f in doc.Descendants("file")
                     select (string)f.Attribute("name");
        lb_filelist.ItemsSource = result;
    }
}
```

Ez a kódrészlet egy tömörített állományban található fájlok listáját tölti le, mely XML formátumban van reprezentálva. Ennek feldolgozását követően a fájlok neveivel feltölti az **lb_filelist** ListBoxot.

Ezek után nézzük meg egy tömörített állomány letöltésének mikéntjét! (A Silverlight csak zippel tömörített állományokat képes kicsomagolni, ezért mi most egy ilyet fogunk letölteni.) A letöltés elindításhoz az **OpenReadAsync** metódusra van szükségünk, míg az eredmény feldolgozásához az **OpenReadCompleted** eseményre. Íme, ezek kódja:

```
private void get_filelist_Click(object sender, RoutedEventArgs e)
{
    ...
    WebClient _client = new WebClient();
    _client.OpenReadCompleted += _client_OpenReadCompleted;
    _client.OpenReadAsync(new Uri("images.zip", UriKind.Relative));
}

void _client_OpenReadCompleted(object sender, OpenReadCompletedEventArgs e)
{
    if (e.Error == null)
        zip_file = new StreamResourceInfo(e.Result, null);
}
```

A letöltött zip fájlt, amely képeket tartalmaz, elmentjük egy privát adattagba **StreamResourceInfo**-ként. Utolsó lépésként nézzük meg, miként lehet egy tömörített állományból kiszedni egy adott fájlt! A ListBox kiválasztott eleme határozza meg, hogy a zipből, az **Application** osztály segítségével, melyik fájlt kell kiolvasnunk. Íme, az ehhez tartozó kód:

```
private void lb_filelist_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    string fname = (string)lb_filelist.SelectedItem;
    StreamResourceInfo image_src = Application.GetResourceStream
        (zip_file, new Uri(fname, UriKind.Relative));
    BitmapImage image = new BitmapImage();
    image.SetSource(image_src.Stream);
    img_file.Source = image;
}
```

A **GetResourceStream** metódus segítségével lehet kiolvasni egy zip fájlból egy tetszőleges erőforrást. Mellesleg a Silverlight is ezt a technikát használja a XAP állományban tárolt fájlok kinyerésére. (Az előbbi

kódrészletben azért volt szükség külön egy **BitmapImage**-re is, ugyanis az Image vezérlő **Source** tulajdonsága nem egy stringet vár értékül, hanem egy **ImageSource**-ot!

Az erőforrások feltöltése egy kicsit bonyolultabb feladat, hiszen ott a fogadóoldali részt is nekünk kell megírni. Íme, egy cikk, amely bemutatja a Silverlightből történő fájlfeltöltést: <http://www.c-sharpcorner.com/UploadFile/nipuntomar/FileUploadsilverlight03182009030537AM/FileUploadsilverlight.aspx>.

Socket programozás Silverlightban

Ehhez a részhez minimális Socket programozási előismeretek szükségesek — legalább annyi, hogy tudjuk, mi az UDP és a TCP, illetve mi a kettő között a különbség (*datagram-socket*, illetve *stream-socket*). Ebben a részben csak a TCP alapú kommunikációval fogunk foglalkozni.

Silverlightban Socket programozás esetén van néhány megkötés, amelyet figyelembe kell vennünk a program írása közben. Az első és legfontosabb megkötés a port tartományra vonatkozik. Ha TCP alapon kommunikálunk, akkor csak és kizárólag a 4502 és a 4534-es közötti portokhoz kapcsolódhat a socket kliens. Egy másik megkötés, hogy szükség van egy policy fájlra is, amelyet a kliens a 943-as porton keresztül kérdez le, ha az és a szerver egy és ugyanazon gépen található. Különben a 80-as HTTP porton keresztül teszi ugyanezt.

UDP esetén az 1024 feletti portok bármelyike használható. Az UDP-nél a policy fájl lekérdezése a 9430-as porton keresztül történik.

Fontos: amennyiben az alkalmazást *Trusted Out Of Browser* módban futtatjuk, akkor nincsenek ezek a megkötések!

Bár elviekben adja magát a dolog, de azért mégis megjegyezem a biztonság kedvéért: *a Silverlight csak kliens lehet a socket modellben*, nem képes bejövő kapcsolatok fogadására (vagyis nem lehet *listener* vagy *server*). Ezért egy mintapéldában a szerver szerepkört egy konzol vagy egy WPF alkalmazásnak célszerű betöltenie. Én az egyszerűség kedvéért egy konzolalkalmazást fogok használni, ami a Socket-es „Hello World” funkciót, vagyis az ún. *EchoServer*-t valósítja meg. Ennek a kódja az alábbi módon néz ki:

```
private static TcpListener listener;
static void Main(string[] args)
{
    listener = new TcpListener(IPAddress.Any, 4502);
    Thread listenToClients = new Thread(new ThreadStart(ListenToClients));
    listenToClients.Start();
}
public static void ListenToClients()
{
    listener.Start();
    while (true)
    {
        Socket socket = listener.AcceptSocket();
        Thread handleClient = new Thread(new ParameterizedThreadStart(HandleClient));
        handleClient.Start(socket);
    }
}

public static void HandleClient(object param)
{
    Socket socket = (Socket)param;
    byte[] bytes = new byte[1024];
    int received = 0;
    while (true)
    {
        try
        { received = socket.Receive(bytes); }
```

```

        catch
        { break; }
        if (received == 0) break;
        Console.WriteLine(Encoding.Unicode.GetChars(bytes, 0, bytes.Length));
    }
    socket.Close();
}

```

Először a program beállítja, hogy bármilyen kliens csatlakozhat hozzá a 4502-es porton, majd elindít egy új szálát (**ListenToClients** metódus), és egy végtelen ciklus segítségével hallgatózni kezd, várja a kapcsolódni kívánó klienseket. Ha van elfogadásra váró socket kliens, akkor elfogadja, és egy új szálban (**HandleClient** metódus) kezeli le. Az új szálban ismét egy végtelen ciklusban hallgatózik, de itt a kliens által küldendő adatokra várva. Ha van bejövő adat, akkor beolvassa azt egy **byte**-tömbbe, visszaalakítja szöveggé, majd végül kiírja a konzolra.

Míg a szervernél a **TcpListener** osztályon volt a hangsúly, addig a kliensnél a **Socket**-en lesz. Silverlight esetén a **Socket** osztály az alábbi fontosabb metódusokkal rendelkezik: **ConnectAsync**, **SendAsync**, **ReceiveAsync**, **Shutdown**, **Close**. (A **ReceiveAsync** függvény duplex — kétirányú — kommunikációnál használatos.) Az első három metódus aszinkron, az utolsó kettő pedig szinkron.

Az EchoServerhez történő csatlakozáshoz a **ConnectAsync** metódust kell meghívni, mely egy **SocketAsyncEventArgs** objektumot vár paraméterül, amelyben meg kell adni a szerver eléréséhez szükséges adatokat. Itt kivételesen ennek kell feliratkozni a **Completed** eseményére, nem pedig a socket-nek, ugyanis annak nincs ilyenje. Íme, tehát a csatlakozás kódja:

```

private Socket socket;
private byte[] bytes;

public MainPage()
{
    InitializeComponent();
}

private void btn_send_Click(object sender, RoutedEventArgs e)
{
    bytes = Encoding.Unicode.GetBytes(tb_message.Text);
    socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
    SocketAsyncEventArgs connectargs = new SocketAsyncEventArgs()
    {
        RemoteEndPoint = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 4502)
    };
    connectargs.Completed += connectargs_Completed;
    socket.ConnectAsync(connectargs);
}

```

A **Completed** eseményben először le kell tesztelnünk, hogy sikeres volt-e a csatlakozás, ezt az **EventArgs** objektumon keresztül könnyen le tudjuk ellenőrizni. Ha sikeres volt, akkor a **SendAsync** metódushoz szükséges **SocketAsyncEventArgs**-ot létre kell hozni és be kell állítani a pufferét, illetve fel kell iratkozni ennek is a **Completed** eseményére. Íme, az adatküldéshez szükséges kód:

```

void connectargs_Completed(object sender, SocketAsyncEventArgs e)
{
    e.Completed -= connectargs_Completed;

    if (e.SocketError != SocketError.Success)
        MessageBox.Show("Hiba lepett fel csatlakozaskor");
    else if (e.LastOperation == SocketAsyncOperation.Connect)
    {
        SocketAsyncEventArgs sendargs = new SocketAsyncEventArgs();
        sendargs.SetBuffer(bytes, 0, bytes.Length);
    }
}

```

```
        sendargs.Completed += (s, ev) => socket.Close();
        socket.SendAsync(sendargs);
    }
}
```

Persze ahhoz, hogy mindez működjön, szükség van egy harmadik programra is, mely a 943-as porton keresztül odaadja a Silverlightos kliensnek a **clientaccesspolicy.xml** fájlt, amely az alábbi módon néz ki:

```
<?xml version="1.0" encoding="utf-8" ?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from>
        <domain uri="*" />
      </allow-from>
      <grant-to>
        <socket-resource port="4502" protocol="tcp" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

Maga a program szintén egy socketes alkalmazás, mely figyelmen kívül hagyja a kérés üzenetét, mindig a policy fájlt küldi vissza. Erre a programra persze csak fejlesztési időben van szükség, éles környezetben a 80-as http porton keresztül kéri le ezt a Silverlight. Íme, a program forráskódja:

```
static void Main(string[] args)
{
    byte[] fileinbytes = File.ReadAllBytes("clientaccesspolicy.xml");
    Socket connectSocket = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Tcp);
    connectSocket.Bind(new IPEndPoint(IPAddress.Any, 943));
    connectSocket.Listen(5);
    byte[] buffer = new byte[1024];
    while (true)
    {
        Socket incomingSocket = connectSocket.Accept();
        incomingSocket.Receive(buffer);
        incomingSocket.Send(fileinbytes, 0, fileinbytes.Length, SocketFlags.None);
        incomingSocket.Close();
    }
}
```

Silverlight kliensek közötti kommunikáció: Local Connection

Nem túl gyakran előforduló feladat, hogy egy oldalon két Silverlight alkalmazást is használnunk kell. Néha azonban akad egy-két olyan eset, amikor nem lehet megkerülni, például: 2 részes hirdetés (egy vízszintes és egy függőleges csík) egy lapon történő megjelenítése miatt, esetleg html overlap elkerülés érdekében. A dolog ott kezd érdekes lenni, amikor ezeknek az alkalmazásoknak adatot kell cserélniük, vagyis kommunikálniuk kell egymással. Ez megoldható a *HTML Bridge* segítségével, de — érezhető — ez nem az igazi. Ezért vezette be a Microsoft már a Silverlight 3-as verziójában a *local connection* vagy *local messaging* néven is emlegetett szolgáltatást, mely megkönnyíti a Silverlight kliensek közötti kommunikációt.

A rendszer rendkívül könnyen használható, mindössze két osztályra van szükségünk, melyek a **System.Windows.Messaging** névtérben találhatóak. Az egyik a **LocalMessageSender**, a másik pedig a **LocalMessageReceiver**. Először a küldés feladatát vizsgáljuk meg.

Első lépésként a **LocalMessageSender** inicializálásakor meg kell adnunk egy egyedi azonosítót, amely majd szűrőként fog szolgálni a fogadó félnél. Vagyis nem a küldő adja meg a címzett nevét, hanem a fogadó szabja meg, hogy kinek az üzenetére kíváncsi. (Így úgymond *broadcast* üzeneteket küldhetünk.) Maga az üzenetküldés a **SendAsync** paranccsal történik, amely használatánál be kell tartanunk két megkötést. Az egyik, hogy csak szöveget lehet átküldeni, a másik, hogy az üzenet maximum 40kb-nyi lehet. Az üzenetküldés kódja ezek után az alábbi módon néz ki:

```
private void btn_send_Click(object sender, RoutedEventArgs e)
{
    LocalMessageSender lm_sender = new LocalMessageSender("LMS");
    lm_sender.SendAsync(tb_message.Text);
}
```

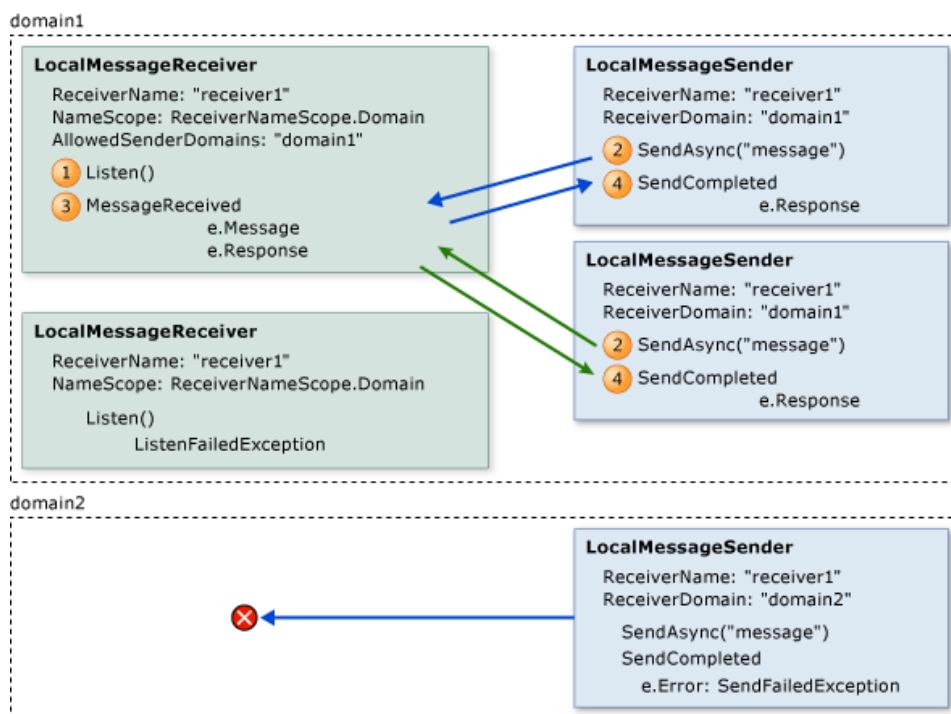
Természetesen feliratkozhatunk a **SendCompleted** eseményre is, amennyiben le szeretnénk ellenőrizni, hogy sikeres volt-e az üzenet elküldése. (Illetve itt lehet feldolgozni az esetleges választ is, lásd később.)

Az üzenetek fogadása sem bonyolult feladat, mindösszesen egyetlen sorral kell többet írni, mint a küldésnél. Először inicializálni kell a **LocalMessageReceiver**-t úgy, hogy megadjuk neki, melyik Silverlight alkalmazástól várjuk az üzeneteket. Utána fel kell iratkozni a **MessageReceived** eseményre, végül pedig meg kell hívni a **Listen** metódust, amely elkezd hallgatózni. Íme, az üzenet fogadásának kódja:

```
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    LocalMessageReceiver lm_receiver = new LocalMessageReceiver("LMS");
    lm_receiver.MessageReceived +=
        (s, ev) => tb_message.Text = ev.Message;
    lm_receiver.Listen();
}
```

Ha szeretnénk egyből választ is küldeni, akkor a **MessageReceived** eseményargumentumának **Response** tulajdonságán keresztül tudjuk ezt megtenni. A küldő oldalon a **SendCompleted** eseménykezelő eseményargumentumának hasonló nevű tulajdonságán keresztül tudjuk elérni a választ.

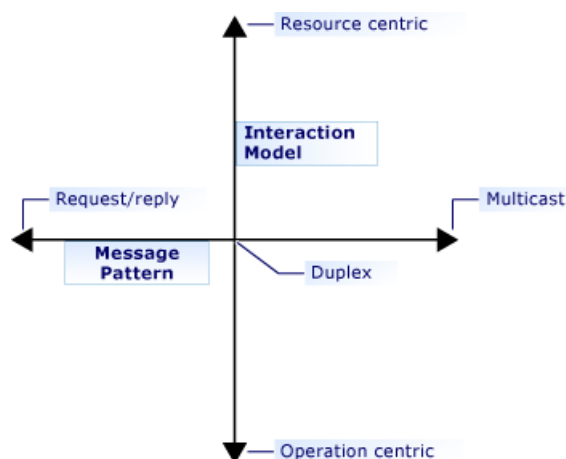
A **LocalConnection**nál arra is van lehetőség, hogy két olyan alkalmazás kommunikáljon egymással, amelyek nem egy tartományon (domainen) belül vannak. Ilyenkor a **Sender**, illetve a **Receiver** objektumok konstruktorának egy másik, túlterhelt változatát kell használni. Ezt a szituációt szemlélteti a 8-35 ábra.



8-35 ábra: Local Connection különböző tartományok között

Összefoglalás

Ebben a fejezetben sokféle szemléletet és elgondolást láthattunk arra, hogy a Silverlight kliens miként tud kommunikálni a szerverrel vagy akár egy másik klienssel, illetve miként lehetséges erőforrásokat kiejánlani. A 8-36 ábra a bemutatott lehetőségeket foglalja össze egy koordináta-rendszerben. Ennek a vízszintes tengelye az üzenetváltás módját, a függőleges tengelye pedig az interakciók módját reprezentálja.



8-36 ábra: Lehetséges kommunikációs irányok

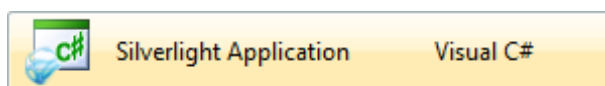
Azt, hogy melyik technológiát mikor érdemes használni, az alábbi oldalon lévő *Deciding which technology to use* rész alatt lévő táblázat foglalja össze: [http://msdn.microsoft.com/en-us/library/cc645029\(v=VS.95\).aspx](http://msdn.microsoft.com/en-us/library/cc645029(v=VS.95).aspx).

9. A Silverlight rejtett képességei

Ebben a fejezetben a Silverlight alapfunktionalitásán túlmutató lehetőségeket mutatom be. Ezt három minialkalmazás segítségével teszem. Egyik sem fog valódi üzleti funkciókat megvalósítani, viszont nagyon hasznos keretként szolgálnak a Silverlight kevésbé szem előtt lévő, „rejtett” képességeinek bemutatásához. Az első alkalmazás neve **SLFeatures** lesz, tartalmát tekintve egyszerű, szerkeszthető tartalmú lista.

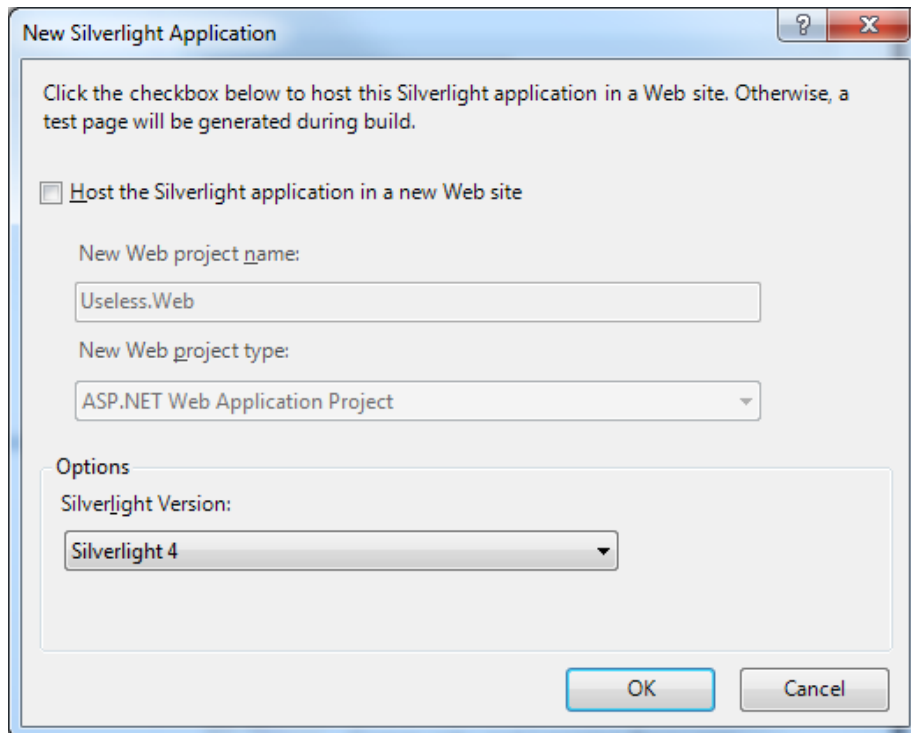
Az alapok összeállítása

Amikor a Visual Studióban új Silverlight alkalmazást hozol létre, több sablon közül is választhatsz. Válaszd a legegyszerűbb sablont, hozz létre egy új C# Silverlight Application projektet (9-1 ábra) **SLFeatures** néven!



9-1 ábra: A kiválasztandó projektsablon

Mivel ebben a fejezetben nem foglalkozunk szerver oldali funktionalitással, a hosztolási kérdésnél válaszd azt a lehetőséget, amelyben a Silverlight alkalmazás hosztíng weboldal nélkül jön létre, amint ezt a 9-2 ábra is mutatja!



9-2 ábra: A Silverlight projektet hosztoló webalkalmazás nélkül hozzuk létre

Ebben az alkalmazásban termékek adatait fogjuk megjeleníteni, szerkeszteni.

A termék adatainak összefoglalására hozz létre a projektben egy új osztályt **Product** néven! Ez az osztály valósítsa meg az **INotifyPropertyChanged** interfészt! Ez az interfész a **System.ComponentModel**

9. A Silverlight rejtett képességei

névtérben található, tehát a kód elején a többi **using** kezdetű sor alatt szükség lesz a névtér beemelésére:

```
using System.ComponentModel;
```

A **Product** osztály tartalmazzon három egyszerű, string típusú tulajdonságot **Code**, **Name** és **Description** néven! Legyen továbbá egy **ImageSource** típusú tulajdonsága **Picture** néven! Ez utóbbi tulajdonság használja az **INotifyPropertyChanged** interfész **PropertyChanged** eseményét:

```
public class Product : INotifyPropertyChanged
{
    public string Code { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }

    private ImageSource picture;
    public ImageSource Picture
    {
        get { return picture; }
        set
        {
            if (picture != value)
            {
                picture = value;
                RaisePropertyChanged("Picture");
            }
        }
    }

    #region INotifyPropertyChanged Members
    public event PropertyChangedEventHandler PropertyChanged;

    void RaisePropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
    #endregion
}
```

Most nem fogunk igazi adatkezelést megvalósítani, de a látszatot azért tartsuk fenn! Készíts egy új osztályt ugyanebbe a fájlba **ProductStore** néven az alábbi kód alapján! Itt a generikus **ObservableCollection** miatt lesz szükséged egy újabb **using** sorra a fájl elején:

```
using System.Collections.ObjectModel;
```

A **ProductStore** osztály kódja legyen az alábbi:

```
public class ProductStore: INotifyPropertyChanged
{
    public ProductStore()
    {
        this.Products = new ObservableCollection<Product>{
            new Product{ Code = "001", Name="Egyeske", Description="Ez egy termék"},
            new Product{ Code = "002", Name="Ketteske", Description="Ez egy másik termék"},
        };
        this.SelectedProduct = this.Products[0];
    }
    public ObservableCollection<Product> Products { get; private set; }
```



```

private Product selectedProduct;
public Product SelectedProduct
{
    get { return selectedProduct; }
    set
    {
        if (selectedProduct != value)
        {
            selectedProduct = value;
            RaisePropertyChanged("SelectedProduct");
        }
    }
}

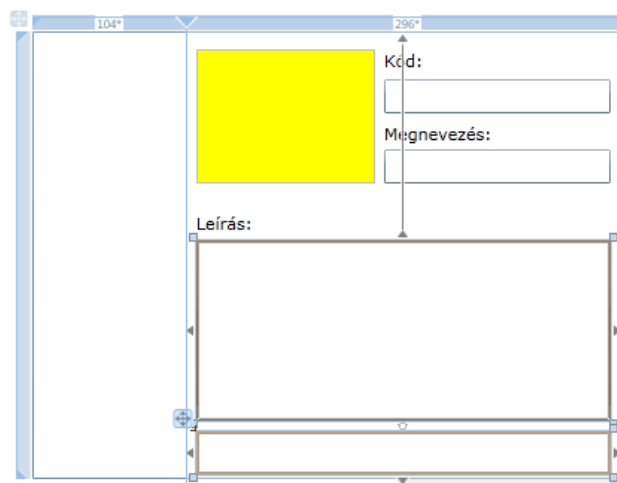
#region INotifyPropertyChanged Members
public event PropertyChangedEventHandler PropertyChanged;

void RaisePropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}
#endregion
}

```

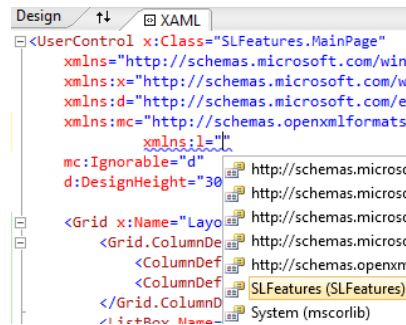
Megjegyzés: A **Product** és **ProductStore** osztályok gyakorlatilag csak adattagokat, tulajdonságokat tartalmazó osztályok. Bizonyos tulajdonságaik azonban nem a legegyszerűbb tulajdonság-definícióval lettek leírva, hanem egy olyannal, amelynek **set** metódusa értesítő eseményt dob az adat változásáról. Ennek az eseménynek a definícióját tartalmazza az implementált **INotifyPropertyChanged** interfész. Ezt az eseményt a felhasználói felület figyeli. További részleteket a 6. fejezetben találsz.

A kapott üres lapon ki fogunk alakítani egy szerkeszthető adatlapot! Oszd két oszlopra a kiindulásként kapott **Grid**-et! A bal oldalra helyezz el egy **ListBox** vezérlőt! Ebben lesznek felsorolva a termékek. A jobb oldalra ízlés szerint helyezz el a termék képének egy **Border** vezérlőt és abban egy **Image** vezérlőt! A **Border** háttérszínét állítsd be valami nem átlátszóra! Helyezz el még két egysoros szövegdobozt (**TextBox**) a termék nevének és kódjának, egy többsoros (**TextBox**, **AcceptReturn=true**) szövegdobozt a termékismertetőnek! Minden szövegdoboz elé tegyél egy-egy, a szövegdoboz tartalmára utaló feliratot (**TextBlock**)! Az adatlap alján pedig a többsoros szövegdoboz alatt helyezz el egy **StackPanel**-t! Ennek a **StackPanel**-nek a tulajdonságlapján az **Orientation** tulajdonságot állítsd át **Horizontál**-ra. Ebbe később gombokat fogsz dobálni. A 9-3 ábra az adatlapot mutatja be szerkesztés közben.



9-3 ábra: A kialakított felhasználói felület

Állítsd be az adatkontextust egy **ProductStore** példányra! Ezt a leggyorsabban a XAML nézetben teheted meg, a **DataContext** blokk beírásával. Ehhez azonban szükséged lesz egy névtérdeklarációra is a **UserControl** xmlns: sorai közt, ahogyan azt a 9-4 ábra mutatja.



9-4 ábra: Az alkalmazás névtérének használata XAML-ben

```
<UserControl x:Class="
...
xmlns:l="clr-namespace:SLFeatures"
mc:Ignorable="d"
d:DesignHeight="300" d:DesignWidth="400" >

<UserControl.DataContext>
<l:ProductStore/>
</UserControl.DataContext>
...
```

Most akár a Visual Studio Property Browser-ével, akár a XAML-be beleírva állítsd be a megfelelő adatkötéseket, például a **ListBox** esetében:

```
<ListBox ItemsSource="{Binding Path=Products}"
SelectedItem="{Binding SelectedProduct, Mode=TwoWay}" DisplayMemberPath="Name" />
```

Vagy a termékkód esetében:

```
<TextBox Text="{Binding SelectedProduct.Code, Mode=TwoWay}" ...
```

A termék képeinek megjelenítéséhez pedig a **Border** belsejében levő **Image** vezérlőnek a következőképp kell kinéznie:

```
<Image Stretch="Uniform" Source="{Binding SelectedProduct.Picture}" />
```

A vágólap kezelése

Ha most elindítod F5-tel az alkalmazást láthatod, hogy működik — ha működésnek lehet nevezni azt, hogy az üres szövegdobozokba azt írhatod, amit csak akarsz. Viselkedj jól nevelt felhasználóként, és próbáld meg egy igazán látszó termék adataival feltölteni a szövegdobozokat! Remélem, átérezed az adatrögzítők sanyarú mindennapjainak súlyát!

Tegyük fel, hogy a helyzetünk sokkal egyszerűbb! A termékek adatait megkaptuk emailben, szövegfájlban vagy Word dokumentumban. Szerencsések vagyunk, hogy a megkapott adatok minden esetben azonos formátumban kerültek begépelésre. Így került előre a termék kódja, majd új sorba a termék neve. A következő sorban pedig a termék leírása kezdődik. Valahogy így:

```
11593
Sóderáj
```

Ebbe a krémbe belefekve és egy-két órát ott töltve lefogyhatsz. Akármilyen undorító is a massa mégiscsak hatásos.

Összetétel:

kénpor 19%

víz 10%

titkos összetevők 71%

Ezt a szöveget már nem kell újra begépelni, egyszerű Ctrl+C és Ctrl+V billentyűkombinációk sorozatával átemelhető az alkalmazás szövegdobozába. Persze ez annyi copy-paste művelet lenne, ahány szövegdoboz van. A rögzített formátum miatt azonban egyszerűen értelmezhető kódból is, hogy melyik bekezdés melyik szövegdobozhoz tartozik. Egy copy-paste művelet is tökéletesen elegendő lenne. A „paste” résszel van csak egy kis gondunk, hiszen az egyszerre csak egy szövegdobozsal működik.

Helyezz el a **StackPanel**-ben egy gombot (**Button**), és változtasd meg a feliratát „Beillesztés”-re, a nevét pedig **CopyButton**-ra! Dupla kattintással iratkozz fel a gomb **Click** eseményére! A **MainPage.xaml.cs** kódban találod magad egy **CopyButton_Click** metódus belsejében. Mielőtt azonban nekiállnál kitölteni, ragadd meg az alkalmat, hogy a deklaratívan elhelyezett adatkontextushoz osztályszintű nevesített referenciát szerezz! Közvetlen az osztálydefiníció alatt definiálj egy tagváltozót!

```
public partial class MainPage : UserControl
{
    private ProductStore store;
    ...
```

Pontosan a definíció alatt találod a konstruktort. Annak a belsejében tárold el a referenciát az adatkontextusra:

```
public partial class MainPage : UserControl
{
    private ProductStore store;

    public MainPage()
    {
        InitializeComponent();
        store = (ProductStore)DataContext;
    }
    ...
```

Ennek a **store** változónak többször is hasznát fogod venni ebben az osztályban. Most töltsd ki az előbb létrejött eseménykezelőt! A **ParseProduct** metódus nevét a szerkesztő pirossal alá fogja húzni — ez egy nem definiált metódus, egyelőre —, de ez ne zavarjon!

```
private void CopyButton_Click(object sender, RoutedEventArgs e)
{
    string productString = Clipboard.GetText();
    Product product = ParseProduct(productString);
    store.Products.Add(product);
    store.SelectedProduct = product;
}
```

A lényeg az első sorban van:

```
string productString = Clipboard.GetText();
```

A **Clipboard** statikus osztály segítségével elérhető az operációs rendszer vágólapja. Két metódus is segít ebben. A **GetText()** a vágólap tartalmát adja vissza szöveges formában (ha ez lehetséges), a

SetText(string) pedig egy szöveget helyez el a vágólapon. Ebből rögtön látszik is, hogy bár elérjük a rendszer vágólapját, de csak egyszerű szöveges adatokkal.

A második sorban használt **ParseProduct** metódust még nem definiáltuk. Ennek semmi más dolga nincs, mint az ismert szöveges formátumot „felaprítani” és egy új **Product** objektumba helyezni. Csak a teljesség kedvéért — nem kevésbé az alkalmazás elindíthatóságának érdekében — írd meg ezt a metódust is!

```
private Product ParseProduct(string productString)
{
    Product parsedProduct = new Product();

    string[] parts =
        productString.Split(new string[] {Environment.NewLine},
            StringSplitOptions.None);

    if (parts.Length > 2)
    {
        parsedProduct.Code = parts[0];
        parsedProduct.Name = parts[1];

        parsedProduct.Description =
            productString.Substring(
                parts[0].Length
                + parts[1].Length
                + Environment.NewLine.Length * 2);
    }
    else
    {
        parsedProduct.Name = productString;
    }
    return parsedProduct;
}
```

A **CopyButton_Click** metódus harmadik sora egyszerűen hozzáadja a terméklistához a **ParseProduct** által előállított objektumot.

```
store.Products.Add(product);
```

Az utolsó sor csak a terméklista megjelenítésének kiválasztását állítja rá az utolsó, tehát az épp hozzáadott elemre, hogy a jobb oldali adatlapon a beállított adatkötések miatt egyből látni is lehessen az eredményt.

```
store.SelectedProduct = product;
```

Próbáld ki az alkalmazást az F5 gomb segítségével! Az elvárt módon formázott szöveg vágólapra helyezése után használd a beillesztés gombot!

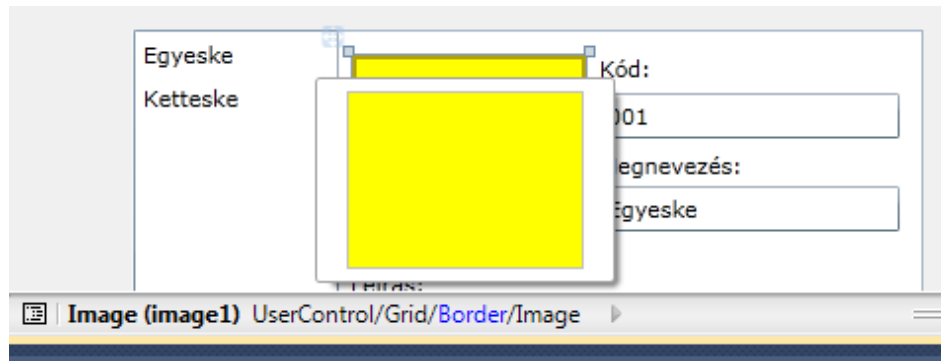
Egy biztonsági figyelmeztetést kapsz. Ez a Microsoft fokozott figyelme a felhasználó érdekeinek védelmében. Legalább egyszer olvasd el! A helyeslő gomb megnyomása után meg kell történnie a terméklista bővítésének.

Ha sikerrel jártál, próbáld meg magad egy új, Másolás feliratú gombbal megoldani az ellenkező irányt. Másold ki az aktuális termék adatait egy jól formázott szövegbe, és helyezd el a vágólapon! Ha nehezen boldogulsz, puskázz a letöltött kódból!

Drag & Drop az operációs rendszerből

Képet nem tudok begépelni. Úgy tippem, hogy te sem. A termék képének megadásához kiválaszthatnánk egy fájlt az erre szolgáló dialógus segítségével. De legyél trendibb! Tedd lehetővé, hogy ebben az alkalmazásban egyszerűen egy könyvtárból lehessen bevontatni a képet! Fájlok behúzását nagyon egyszerűen megoldhatod, és az itt használt képbeolvasáson túlmenően bármilyen adatot is megszerezhetsz ezzel a módszerrel.

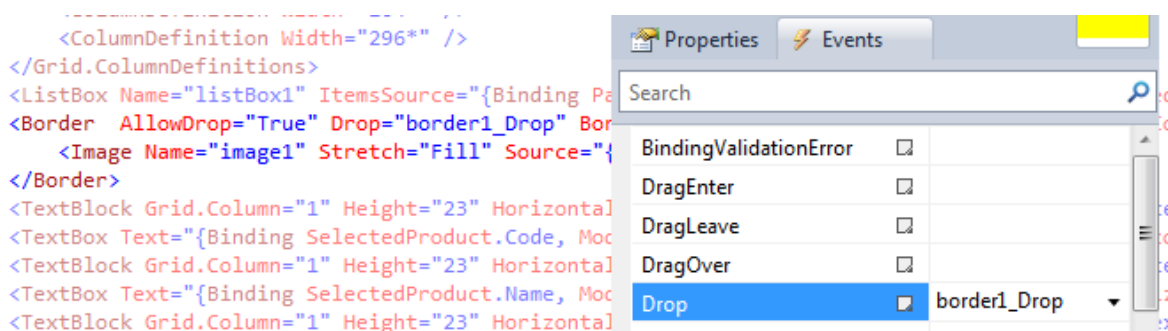
Itt használd ki a termék képe köré rakott **Border** vezérlőt. Keresd meg — a benne terpeszkedő **Image** vezérlő miatt segíthet a szerkesztő felület alján található morzsa (9-5 ábra) —, és állítsd be az **AllowDrop** tulajdonságát **true**-ra!



9-5 ábra: A Border vezérlő megkeresése az Image alatt

Ezzel a tulajdonsággal „kéred meg”, hogy fogadja el a rádobott adatokat. Ezt természetesen be lehetne állítani a **UserControl**-on is. Abban az esetben mindegy lenne, hogy az alkalmazás mely részére dobjuk a fájlt. A **Border** használata viszont megmutatja, hogy tetszés szerint lehet kijelölni egy vagy akár több fogadó területet is.

Már csak egy dolgod van, a **Drop** eseményre való feliratkozás. Ezt a Visual Studio Property Browserében az események közt teheted meg a legegyszerűbben. De ha úgy tartja kedved, a 9-6 ábrán látható **Drop="border1_Drop"** attribútumot kézzel is beírhatod. Ebben az esetben az idézőjelek közötti részen kell a jobb egérgombbal kattintanod, hogy a szerkesztőfelület a **Navigate to Event Handler** menüponttal létrehozza a metódust és rögtön oda is navigáljon.



9-6 ábra: Feliratkozás a Border vezérlő Drop eseményére

A **Drop** eseménykezelő metódusában meg kell írni a fájlrendszerből „bedobott” fájlok felhasználását. Ez önmagában nem bonyolult, de mi belekeverünk egy kis file kezelést is.

Először írd meg egy **if** blokkot, amiben ellenőrzöd, hogy jött-e adat:

```
private void border1_Drop(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
    {
```

```
}  
}
```

Az **if** blokk belsejében egy kódsorral megszerezheted az összes bedobott fájl adatait. A **FileInfo** használatához szükség lesz a fájl elején a **using System.IO** sorra.

```
private void border1_Drop(object sender, DragEventArgs e)  
{  
    if (e.Data.GetDataPresent(DataFormats.FileDrop))  
    {  
        FileInfo[] files = e.Data.GetData(DataFormats.FileDrop) as FileInfo[];  
    }  
}
```

A **GetData** metódus egy tömböt ad vissza, amelynek az elemein egy **foreach** ciklussal végig lehet menni — ha egyszerre több fájlt vontat valaki a képeretünkre. Mivel a jelenlegi megoldásunkhoz csak egyetlen képfájlról van szükség, egyszerűen használd a megkapott tömb legelső elemét! Ellenőrizd le, megfelelő formátumú-e! Az egyszerűség kedvéért ezt most csak a fájl kiterjesztésének ellenőrzésével végezd:

```
private void border1_Drop(object sender, DragEventArgs e)  
{  
    if (e.Data.GetDataPresent(DataFormats.FileDrop))  
    {  
        FileInfo[] files = e.Data.GetData(DataFormats.FileDrop) as FileInfo[];  
  
        if (files[0].Extension == ".png" || files[0].Extension == ".jpg")  
        {  
        }  
    }  
}
```

Ha ez is rendben van, akkor a **FileInfo** által reprezentált fájlt be kell olvasni és el kell tárolni az aktuálisan megjelenített termék **Picture** tulajdonságában.

```
private void border1_Drop(object sender, DragEventArgs e)  
{  
    if (e.Data.GetDataPresent(DataFormats.FileDrop))  
    {  
        FileInfo[] files = e.Data.GetData(DataFormats.FileDrop) as FileInfo[];  
  
        if (files[0].Extension == ".png" || files[0].Extension == ".jpg")  
        {  
            BitmapImage img = new BitmapImage();  
            using (Stream s = files[0].OpenRead())  
            {  
                img.SetSource(s);  
            }  
            store.SelectedProduct.Picture = img;  
        }  
    }  
}
```

Futtasd az alkalmazást az F5-tel, és próbáld ki, mi történik, ha képeket dobsz rá!

Megjegyzés: A **BitmapImage** rendelkezik egy olyan konstruktorral is, aminek rögtön át lehet adni **Uri** típusként a kép elérhetőségét. Ha ez sikerül, akkor nem kell neked **Stream**-et nyitni. Ebben az esetben

azonban a Silverlight szigorú biztonsági előírásai miatt a **FileInfo** objektum **FullName** tulajdonsága nem elérhető. Ez szándékosan van így, hogy ne lehessen garázdálkodni a gyanútlan felhasználó merevlemezén.

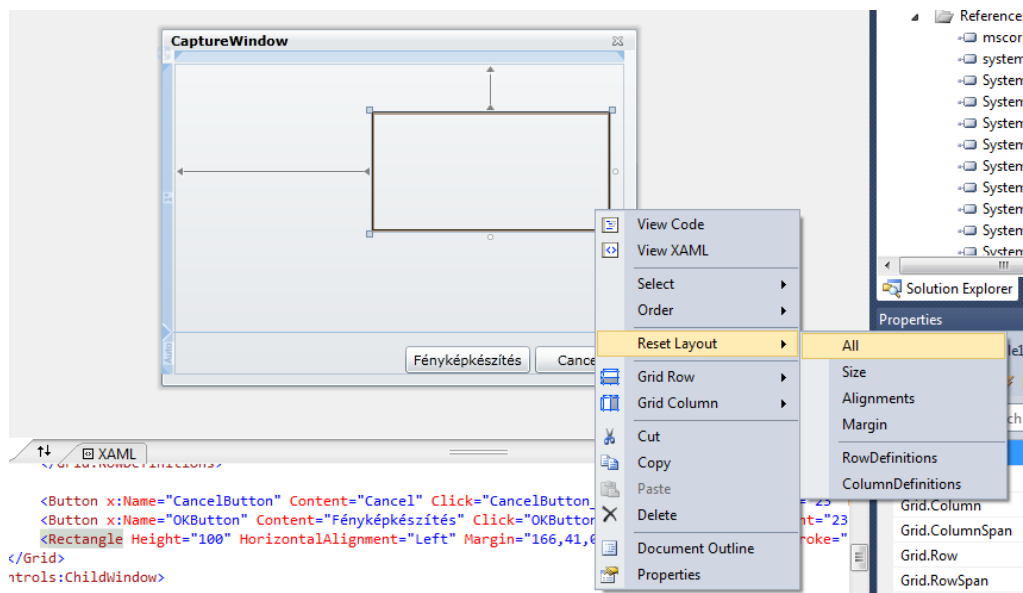
Kamerakezelés

Látványos konferenciákról nem hiányozhat a videó. Ha ez élő videó, akkor még érdekesebb. Ezért már biztosan mindenki tudja, hogy a Silverlight 4 is képes kezelni a webkamerákat és a mikrofonokat. A szokásos példaalkalmazás vagy egy fényképkészítő vagy egy videó konferencia megvalósítása szokott lenni. De egy kis fantáziával sok hasznos dologra is használhatóvá válik ez az input lehetőség.

Ebben az alkalmazásban azt az esetet szimuláljuk, amikor egy termékhez még nincs kész képünk, hanem szeretnénk azt lefotózni. Ha van kéznél webkamerád, akkor kalandra fel! A „sóderáj” képének elkészítéséhez hozz a fürdőszobából valami felismerhetetlen krémet!

Ennek a műveletnek a leprogramozásához egy kódsornál többre lesz szükség, de megijedni nem kell. Először a felületet kell egy kicsit módosítani, hogy legyen egy terület a kamera képének megjelenítéséhez. Erre nem alkalmas a már felhasznált **Image** vezérlő, hiszen nem állóképről van szó. Sőt, nincs is külön erre a célra semmilyen vezérlő! A webkamera képét egy különleges ecseten (**Brush** objektum) keresztül láthatjuk, és ezzel szinte bármit kifesthetünk! Jelen esetben erre a célra egy egyszerű téglalapot használj — az érdekesség kedvéért más alakzatokkal is kipróbálhatod.

A fényképezéshez készíts egy külön ablakot! Adj a projekthez egy új **ChildWindow** elemet **CaptureWindow** néven! Ez a sablon már tartalmaz két gombot. Az OK gomb feliratát változtasd meg „Fényképkészítés”-re, és töröld a szélességét, hogy beleférjen az új szöveg! A maradék helyre a Toolboxról húzz be egy **Rectangle** vezérlőt — ebben a téglalapban fog megjelenni a kamera képe! Hogy teljesen kitöltse a rendelkezésre álló területet, a téglalapon jobb-kattintva megjelenő menüben válaszd ki a **ResetLayout** és azon belül az **All** menüpontot (9-7 ábra)! Ellenőrizd le a téglalap nevét, mert arra a későbbiekben szükségünk lesz — **rectangle1**-nek kell lennie!



9-7 ábra: A téglalap teljes méretűre állítása

Most az ablak képén bárhol jobb-kattintva válaszd ki a **View Code** menüpontot, hogy a C# kódba kerülj! Az ablak kódjában két osztályszintű elemre lesz szükség. Egyrészt egy tulajdonságra, amiben a webkameráról beolvasott képet tárolhatod. Másrészt arra az eszközre, ami segít azt a képet megszerezni. A képet ún. **WritableBitmap** formában fogod megkapni, a műveletet pedig a **CaptureSource** osztály segítségével lehet elvégezni. A kód eleje így nézzen ki:

```
public partial class CaptureWindow : ChildWindow
{
```

```
public WriteableBitmap CapturedImage { get; private set; }
CaptureSource source = new CaptureSource();
...
```

Mivel ennek az ablaknak egyetlen célja a **CaptureSource** példány használata, ezért a konstruktorban el is végezheted a munka dandárját. Az **InitializeComponent** hívás után fel kell konfigurálni a videó forrást. Ennek legegyszerűbb módja az alapértelmezett videó egység használata. Írd tehát a konstruktorba a következőket:

```
public CaptureWindow()
{
    InitializeComponent();
    source.VideoCaptureDevice =
        CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
    source.CaptureImageCompleted += source_CaptureImageCompleted;
}
```

Az első beírt sor egyszerűen elkéri a rendszertől az alapértelmezett videó forrást. A következő sor arra az eseményre iratkozik fel, ami egy fénykép elkészülésekor fog bekövetkezni. Ezt az eseménykezelő metódust is hozd létre, egyelőre csak üresen!

```
void source_CaptureImageCompleted(object sender, CaptureImageCompletedEventArgs e)
{
}
```

A következő feladat, hogy az ablakon elhelyezett téglalapban megjelenítsd a kamera élő képét. Erre fogod használni a **VideoBrush** nevű ecsetet. A Silverlightban a vezérlők kiszínezésére, kifestésére ecseteket használhatsz. Ez egy olyan különleges ecset, ami nem egy színnel, de még csak nem is színátmenettel, hanem mozgóképpel tölti ki a színezendő felületet. A mozgókép két forrásból érkezhetsz. Eredetileg a **MediaElement** tartalmát lehetett csak vele festésre használni, a négyes verzió óta azonban a videó forrás élő képét is. Természetesen, ezt az utóbbit fogjuk választani.

Először szükség lesz egy **VideoBrush** példányra. Állítsd be, hogy a festésre használandó képet milyen módon méretezze a festendő terület méretére! Ezt a **Stretch** tulajdonságával teheted meg. Állítsd be azt is, hogy honnan szerezze a videót a **SetSource** metódus meghívásával és a **CaptureSource** példány átadásával! Végül, de nem utolsósorban az elkészült **VideoBrush** példányt add meg a téglalap kiszínezéséhez annak **Fill** tulajdonságával!

```
public CaptureWindow()
{
    InitializeComponent();
    source.VideoCaptureDevice = CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
    source.CaptureImageCompleted += source_CaptureImageCompleted;

    VideoBrush brush = new VideoBrush();
    brush.Stretch = Stretch.Uniform;
    brush.SetSource(source);
    rectangle1.Fill = brush;
}
```

A feladat lényegi részével készen is vagy. Kipróbálásához azonban még néhány apróság hiányzik! Először is a biztonsággal kapcsolatos beállítások. Weblapokon elhelyezett Silverlight alkalmazások a számítógépedhez csatlakoztatott webkamerát elindítva észrevétlenül kémkedhetnének utánad! Ez, ugye, nem hangzik biztatóan? Éppen ezért, mielőtt a kamerát használni kezdenéd, engedélyt kell kérni a felhasználótól. Ezt a **CaptureDeviceConfiguration** objektum **RequestDeviceAccess** metódusával teheted meg. Ezt a kérdést azonban egyfolytában feltenni nagyon zavaró, tehát ha egyszer már

megengedte a felhasználó, többé fölösleges zaklatni. Ezt az állapotot tükrözi az **AllowedDeviceAccess** tulajdonság. A kamera használatának elindítása pedig a **CaptureSource** példány **Start** metódusával történik. Ennek megfelelően, az alábbiakkal egészítsd ki a konstruktor kódját:

```
public CaptureWindow()
{
    InitializeComponent();
    source.VideoCaptureDevice = CaptureDeviceConfiguration.GetDefaultVideoCaptureDevice();
    source.CaptureImageCompleted += source_CaptureImageCompleted;

    VideoBrush brush = new VideoBrush();
    brush.Stretch = Stretch.Uniform;
    brush.SetSource(source);
    rectangle1.Fill = brush;

    if (CaptureDeviceConfiguration.AllowedDeviceAccess
        || CaptureDeviceConfiguration.RequestDeviceAccess())
    {
        source.Start();
    }
    else
    {
        this.DialogResult = false;
    }
}
```

Az **else** ágban található **DialogResult** beállítás a Silverlight ablak bezárását eredményezi. Bár valójában fényképet még nem készít a kód, de már elég látványos lenne, ha ezt az újonnan létrehozott ablakot elő tudná csalogatni a felhasználó! E célból most egy kicsit térj vissza a **MainPage**-re és helyezz el a Másolás és Beillesztés gomb mellé egy újabb gombot „Fényképkészítés” felirattal és **TakePictureButton** néven. Ez a gomb fogja megmutatni az előbb elkészített fényképező ablakot. A **Click** eseménykezelőjébe írd be az alábbi kódot:

```
private void TakePictureButton_Click(object sender, RoutedEventArgs e)
{
    CaptureWindow cw = new CaptureWindow();
    cw.Show();
}
```

Pihenésképpen nyomd meg az F5 gombot, és próbáld ki az eddig elkészülteket!

Most következik a fényképkészítés! A fényképkészítő ablakon (**CaptureWindow**) két gomb van. A **Cancel** gomb megnyomásakor nem szabad képet készíteni, a **Fényképkészítés** gomb megnyomásakor viszont kötelező. Ha készült kép, akkor az aktuálisan kiválasztott termék képét azzal kell frissíteni.

A legutóbb a **MainPage** kódjába írt **TakeAPictureButton_Click** eseménykezelőt egészítsd ki azzal, hogy észlelni tudja a **CaptureWindow** bezárását és azt is, hogy készült-e kép vagy sem:

```
private void button3_Click(object sender, RoutedEventArgs e)
{
    CaptureWindow cw = new CaptureWindow();
    cw.Closed += delegate
    {
        if (cw.DialogResult == true)
        {
            store.SelectedProduct.Picture = cw.CapturedImage;
        }
    };
    cw.Show();
}
```

A kiemelt kódrészlettel feliratkozhat a **CaptureWindow** példány **Closed** eseményére. Ha a felhasználó bezárja az ablakot – akármelyik gombbal is történjen az –, ez az eseménykezelő fog végrehajtódni. A kiemelt kódrészlet a harmadik sorától fog végrehajtódni, ha az ablakot bezárták.

Az eseménykezelő a bezárt ablak **DialogResult** értékétől függően beállítja az aktuálisan kiválasztott termék **Picture** tulajdonságát arra, ami az ablak **CapturedImage** tulajdonságában található.

Az ablak **DialogResult** tulajdonságát a **CaptureWindow** kódjában kell beállítani. Az értékadás után az ablak egyből bezáródik. Váltás át a **CaptureWindow** C# forráskódjára, és nézd meg a két utolsó metódust!

Az **OKButton_Click** és a **CancelButton_Click** eseménykezelők mindegyike a **DialogResult** tulajdonságnak ad értéket (ezzel bezárva az ablakot), csak épp az OK a **true**, míg a Cancel a **false** értéket.

A Cancel gomb megnyomásakor vége a dalnak. Hogy szemetet ne hagyj magad után, a **DialogResult** beállítása előtt még állítsd le a **CaptureSource** működését!

```
private void CancelButton_Click(object sender, RoutedEventArgs e)
{
    source.Stop();
    this.DialogResult = false;
}
```

Az OK gomb esetében azonban van egy kis plusz feladatod – meg kell szerezni a képet. Az OK gomb **Click** eseménykezelőjét írd át, hogy az a **CaptureSource** objektumtól kérje el az aktuális videóképet! Töröld ki belőle a **DialogResult** beállítását, mert nem szeretnénk azonnal bezárni az ablakot, csak ha megvan a kép!

```
private void OKButton_Click(object sender, RoutedEventArgs e)
{
    source.CaptureImageAsync();
}
```

A hívott metódus nevéből lehet következtetni arra, hogy a kép megszerzése nem azonnal történik. Még a konstruktorban feliratkoztál a **CaptureSource** példány (**source** változó) **CaptureImageCompleted** eseményre. Az ahhoz létrehozott eseménykezelő fog meghívódni, ha a **CaptureImageAsync** metódus végzett. Keresd meg az akkor létrehozott üres eseménykezelő metódust! Az eseménykezelőben mentsd el a képet a **CapturedImage** tulajdonságba, majd állítsd le a **CaptureSource**-t, és végül a **DialogResult** **true**-ra állításával zárd be az ablakot:

```
void source_CaptureImageCompleted(object sender, CaptureImageCompletedEventArgs e)
{
    CapturedImage = e.Result;
    source.Stop();
    this.DialogResult = true;
}
```

Ezzel a fényképkészítés funkció elkészült, ki lehet próbálni!

Teljes képernyő használata

Egy munkahely kialakításánál fontos lehet, hogy a képernyőn csak a munkához szükséges eszközök legyenek, azoknak viszont a lehető legtöbb hely jusson. A felhasználó maximalizálhatja a böngésző ablakát, de néha az ablak így megmaradó területe is kevés lehet. Szeretnénk több dologtól is megszabadulni, mint például a Windows tálcájától vagy az ablakunk még így is felfedezhető szélétől: teljes képernyőt szeretnénk! Ezt a következő kódsorral lehet megtenni:

```
Application.Current.Host.Content.IsFullScreen = true;
```

A kipróbálásához a szokásos gombsor végére helyezze el egy újabb gombot „Teljes képernyő” felirattal és **FullScreenButton** névvel! A gomb **Click** eseménykezelőjébe írja be a fenti sort és próbálja ki!

Két említésre méltó dolgot kell tapasztalnod:

Egyrészt a teljes képernyőre váltáskor a Silverlight figyelmezteti a felhasználót, hogy az alkalmazás átáll teljes képernyős módba. Felhívja továbbá a figyelmet az Esc gomb hasznosságára, amivel meg lehet szüntetni a teljes képernyős módot. Az Esc gomb működése miatt nincs is szükség a teljes képernyős módot kikapcsoló funkció megírására. Nem mintha bonyolult lenne, hiszen az előbb megismert tulajdonság értékét **true** helyett **false**-ra állítva pontosan ez történik. A képernyőmód váltására figyelmeztetéstől és az Esc gomb funkciójától nem lehet megszabadulni — a felhasználó biztonsága érdekében. Ha ez nem így lenne, akkor nagyon egyszerűen lehetne egy weblapra navigálás után automatikusan teljes képernyős módba váltva az operációs rendszer képernyőjét szimulálva — például egy jelszót bekérő ablak megjelenítésével — jelszót lopni. Ez néha még ilyen trükkök nélkül is sikerülhet a tájékozatlan és jóhiszemű felhasználók esetében.

A második említésre méltó dolog az, hogy teljes képernyős módban nem működnek a szövegdobozok — nincs teljes billentyűzet támogatás — szintén biztonsági megfontolásból. Ez rengeteg alkalmazás működését lehetetlenné teszi. Felmerülhet a kérdés, hogy akkor mire is jó egyáltalán ez az üzemmód? Leggyakoribb felhasználása videók lejátszása. Ha üzleti alkalmazást szeretnénk ebben a módban használni, arra is van lehetőségünk a hamarosan tárgyalt *Out-Of-Browser* (OOB) technológia segítségével. Böngészőn kívül futtatott Silverlight alkalmazással több dolgot megtehetünk, mint egy böngésző belsejében. Ilyen esetben a teljes képernyős figyelmeztetés és az Esc gomb funkciója is letiltható, ekkor nekünk kell megírni a teljes képernyős módot kikapcsoló funkciót is. Cserébe azonban működhetnek a szövegdobozaink. Ehhez az OOB *Elevated Trust* módja szükséges (lásd később).

Nyomtatás

Ez olyan téma, aminek a fontosságáról senkit sem kell meggyőzni. Ez annyira igaz, hogy a korábbi Silverlight változatoknál az első kérdések egyike mindig a nyomtatás lehetőségére vonatkozott. Akkoriban az „alapvetően, nem lehet” választ többnyire szemérmesen elkerültük. Úgy érzem azonban, hogy érdemes volt várni, mert ennél egyszerűbben kezelhető, mégis minden helyzetben használható megoldást nem is kaphattunk volna!

A nyomtatandó látványt a szokásos UI tervező eszközökkel megtervezheted. Használhatsz adatkötést és a nyomtatás előtti pillanatban beállíthatod az adatkontextust és a méretet is. Nem kell vonalakat, téglalapokat, képeket vagy egyéb grafikus elemeket kódból rajzolgatni!

Adj a projekthez egy új **UserControl** objektumot, **PrintPage** néven! Ez lesz a nyomtatandó tartalom. Az üres felületet méretezd át A4-es arányúra (nem kell pontosan, elég nagyjából, hiszen a XAML biztosíthatja a gumifelületet)! Ízlés szerint rendezd be egy termék adatait, képét megjelenítő vezérlőkkel! Ügyelj arra, hogy lehet, teljesen más méretben lesz nyomtatva — használd ki a **Grid** oszlopai és sorai által adott automatikus elrendezés lehetőségeit, vagy használd a **ViewBox** vezérlőt! Minden vezérlőnek állítsd be az adatkötését! Ezt egyszerű esetekben, mint amilyen ez a példaalkalmazás, akár kézzel is beírhatod a XAML-be. Ez most egy egyszerű eset, tehát az alábbi kódrészlet pont megfelel a **PrintPage.xaml** belsejének.

```
<Viewbox Stretch="Uniform" VerticalAlignment="Top">
  <Grid x:Name="LayoutRoot" Background="White" Width="300">
    <Grid.RowDefinitions>
      <RowDefinition Height="auto" />
      <RowDefinition />
      <RowDefinition Height="auto" />
    </Grid.RowDefinitions>
    <TextBlock Text="{Binding Name}" FontSize="25" />
    <Image Source="{Binding Picture}" Grid.Row="1" Width="100" />
    <TextBlock Text="{Binding Description}" TextWrapping="Wrap" Grid.Row="2" FontSize="10"/>
  </Grid>
</Viewbox>
```

Helyezz a **MainPage**-re egy újabb gombot Nyomtatás felirattal és **PrintButton** néven! Nagyjából itt fogyhatott el a hely a szerkesztő felületeden — nyugodtan szélesítsd az egész **MainPage**-t! Dupla kattintással iratkozz fel a gomb **Click** eseményére és alakítsd át a következő módon:

A metódus előtt, közvetlen az osztály törzsében definiálj egy **PrintDocument** típusú változót **printer** néven! Az eseménykezelő belsejében hozz is létre egy ilyen objektum példányt, azt a feltételt kezelve, hogy a **printer** változóban még nincs semmi. A létrehozás után rögtön iratkozz fel a **PrintPage** eseményre. Ezzel biztosítottad, hogy mindig legyen printer objektumod, meghívhatod annak **Print** metódusát. A **Print** metódus egyetlen szöveget vár paraméterül, ez fog a nyomtatási sorban a dokumentum neveként szerepelni.

```
PrintDocument printer;
private void PrintButton_Click(object sender, RoutedEventArgs e)
{
    if (printer == null)
    {
        printer = new PrintDocument();
        printer.PrintPage += new EventHandler<PrintPageEventArgs>(printer_PrintPage);
    }

    printer.Print(store.SelectedProduct.Name);
}
```

Ha alaposan átnézed a kódot, feltűnhet, hogy nagyon fontos dolgoknak nyoma sincs. Melyik nyomtatóra szeretnénk nyomtatni? Mi az, amit ki szeretnénk nyomtatni?

Az első kérdésre kapjuk meg a legegyszerűbben a választ. Kivételesen a biztonsági elvárások összhangban vannak az egyszerűséggel. A **Print** metódus minden hívásakor a felhasználó az operációs rendszer nyomtatóválasztó ablakával találja szemben magát. Amelyik nyomtatót kiválasztja, arra fog az alkalmazás nyomtatni. Nincs lehetőség kódból ráerőltetni egyik nyomtatót sem!

A „mit nyomtatunk” kérdésre pedig a **PrintPage** eseménykezelőjében adhatod meg választ.

```
void printer_PrintPage(object sender, PrintPageEventArgs e)
{
    FrameworkElement pageToPrint = new PrintPage();
    pageToPrint.DataContext = store.SelectedProduct;
    pageToPrint.Width = e.PrintableArea.Width - e.PageMargins.Left - e.PageMargins.Right;
    pageToPrint.Height = e.PrintableArea.Height - e.PageMargins.Top -
        e.PageMargins.Bottom;

    e.PageVisual = pageToPrint;
}
```

A metódustörzs első négy sora létrehoz egy **PrintPage** példányt (ez az a UserControl, amelyen összeállítottad a nyomtatandó dokumentum képét). Adatkontextusnak az éppen kiválasztott terméket állítja be; az esemény paraméterein keresztül megszerzett nyomtatási információk alapján beállítja a lap méretét a nyomtató által várt méretre.

Az utolsó sor pedig az esemény paraméterén keresztül elmondja a Silverlightnak, hogy ezt a vezérlőt jelenítse meg nyomtatásban.

Megjegyzés: Ebben a példában egy oldal nyomtatásához mindig egy új vezérlőt példányosítottál. Ez nem kell, hogy mindig így legyen. Több oldal nyomtatásakor vagy nyomtatás intenzív alkalmazása esetén dolgozhatsz előre is (például használhatsz deklaratív adatkontextust a nyomtatandó vezérlőn, osztályszinten létrehozhatsz egy példányt a nyomtatandó vezérlőből, és az adatkötés dinamizmusát kihasználva egyfolytában újrahasználhatod azt).

További két kérdést még meg kell válaszolni. Hogyan nyomtathatunk több oldalon keresztül? Mi történik nyomtatási hiba esetén?

A többoldalas nyomtatás nem automatikus! Ezt meglehetősen nehéz automatizálni, hiszen több módja is van a sok oldal leképezésének. Lehet, hogy minden oldal teljesen külön életet él, például ha egyszerre több terméket szeretnénk nyomtatni, de mindegyiket külön lapra. Egy hosszú adatlista, táblázat esetében azonban hasznos, ha minden oldal tetején megismétlődik a táblázat fejléce. Van, amikor szeretnénk sorszámokat is megjeleníteni, és van, amikor nem. Rengeteg kívánságunk lehet többoldalas nyomtatás esetén! Márpedig aki ennyire rigolyás, az oldja meg maga!

Megjegyzés: Ha olyan sok nyomtatnivalód akad, ami csak több oldalra fér el, meg kell állapítani, mettől meddig fér az első, második stb. oldalakra a tartalom. Erre sajnos a próbálgatásos módszer a legalkalmasabb. Ha például a nyomtatandó **UserControl LayoutRoot** nevű **Grid** vezérlőjét **StackPanel**-re cseréled, és abba csak egy **TextBlock**-ot helyezel el, akkor a **TextBlock** szövegének beállítása után kiolvashatod annak magasságát. Ha az nagyobb, mint a nyomtatási terület magassága, akkor kevesebb szöveggel újra megpróbálhatod, egészen addig, amíg bele nem fér az oldalba. Ezzel a szöveggel továbbbengedheted, persze előtte meg kell jegyezni, hogy meddig sikerült belepakolni a nyomtatandó tartalmat, és be kell állítani, hogy még további oldalakat is nyomtatni kívánsz. A következő oldalnál azzal a tartalommal folytathatod a munkát, ami az előbbi oldalba már nem fért bele.

A fejezet példakódjában elhelyeztem egy többoldalas nyomtatás mintát is.

A Nyomtatás gomb **Click** eseménykezelőjében iratkoztál fel a **PrintPage** eseményre. Ezenkívül még kettő – a **BeginPrint** és az **EndPrint** – eseményekre lehet feliratkozni.

```
PrintDocument printer;
private void PrintButton_Click(object sender, RoutedEventArgs e)
{
    if (printer == null)
    {
        printer = new PrintDocument();
        printer.PrintPage += new EventHandler<PrintPageEventArgs>(printer_PrintPage);
        printer.BeginPrint += new EventHandler<BeginPrintEventArgs>(printer_BeginPrint);
        printer.EndPrint += new EventHandler<EndPrintEventArgs>(printer_EndPrint);
    }

    printer.Print(store.SelectedProduct.Name);
}
```

A **BeginPrint** eseménykezelő azon a tényen kívül, hogy elkezdődött a nyomtatás, más értékes információt nem szolgáltat. Az **EndPrint** eseményben viszont az átadott esemény-argumentumban van egy **Error** tulajdonság. Ha nyomtatás közben valami hiba történt, arról ezen a tulajdonságon keresztül lehet információt szerezni.

```
void printer_EndPrint(object sender, EndPrintEventArgs e)
{
    if (e.Error != null)
    {
        //hiba történt nyomtatás közben
    }
}
```

Összefoglalva:

1. Egy **PrintDocument** példányon meghívjuk a **Print** metódust.

2. Bekövetkezik a **BeginPrint** esemény. Itt értesül a kódunk a nyomtatás indításáról. Itt például le lehet tiltani a Nyomtatás gombot, ki lehet írni egy státusz üzenetet, stb.

3. Bekövetkezik a **PrintPage** esemény. Itt kell szereznünk vagy összeállítanunk egy **UserControl** példányt, amelyen minden kinyomtatandó információt megjelenítettünk (alapvetően adatkötés segítségével). Illik ennek a **PrintPage** példánynak a méretét arra beállítani, amit az esemény paraméteréből kiolvashatunk — hiszen különböző méretű papírokra nyomtathat a felhasználó. Ezt a **UserControl** példányt az eseményparaméter **PageVisual** tulajdonságába kell beírni.

És itt jön egy extra lépés! Az eseményparaméter **HasMorePages** tulajdonságát **false** értékkel kapjuk meg. Ha azonban az oldal összeállítása közben úgy döntenénk, hogy lesz még folytatás, át kell állítani **true**-ra.

4. Ha a **HasMorePages** értéke **true**, akkor újra a 3. lépés jön.

5. Bekövetkezik az **EndPrint** esemény. Vége a nyomtatásnak.

Hálózatészlelés

A Silverlight hálózati alkalmazás, ezért futása közben is szüksége lehet a hálózat elérésre, illetve annak a ténynek az ismeretére, hogy egy adott pillanatban a hálózat éppen elérhető-e vagy sem. A **NetworkInterface** objektum segítségével figyelheted az elérhető hálózatok állapotváltozásait, így lehetőség nyílik a hálózat elvesztésének érzékelésére — hibaüzeneteket megelőzve —, illetve a hálózati kapcsolat helyreállásakor, felépülésekor azonnal elkezdheted az élő kapcsolatot feltételező tevékenységet.

Ehhez mindössze két statikus osztályra van szükséged a **System.Net.NetworkInformation** névtérből:

- A **NetworkInterface** osztály egyetlen hasznos metódussal rendelkezik: **GetIsNetworkAvailable**, ez a név önmagáért beszél. Egy **bool** értékben visszaadja, hogy épp csatlakozva vagyunk-e valamilyen hálózathoz vagy sem.
- A **NetworkChange** osztály egyetlen eseménnyel rendelkezik. Erre feliratkozva értesítést kaphatunk a hálózat változásairól. Az esemény neve: **NetworkAddressChanged**.

Nézzük meg ezek használatát a gyakorlatban is! Nyisd meg a **MainPage.xaml.cs** fájlt és a **MainPage** osztály belsejében hozz létre egy metódust **UpdateNetworkStatus** néven! Ennek törzsében állítsd át a **LayoutRoot** háttérszínét a hálózat aktuális állapotától függően az alábbi kódrészlet segítségével:

```
private void UpdateNetworkStatus()
{
    bool isNetwork = NetworkInterface.GetIsNetworkAvailable();
    if (!isNetwork)
        this.LayoutRoot.Background = new SolidColorBrush(Colors.Red);
    else
        this.LayoutRoot.Background = new SolidColorBrush(Colors.Green);
}
```

Fontos megfigyelni, hogy a **GetIsNetworkAvailable** metódus nem vár paramétert! Ebből már látható, hogy több hálózati kapcsolat esetén nem tudod azokat megkülönböztetni. Elég, ha az egyiken van élő kapcsolat, és máris **true**-val jutalmaz. Továbbá nem ellenőrzi, hogy az a hálózati kapcsolat értelmes kapcsolat-e egyáltalán. Rossz IP címmel, használhatatlan átjáróval ugyanúgy **true**-t kapsz vissza. Ha viszont egyik hálózati kártyában sincs beillesztett vezeték (már ha vezetékes hálózatról van egyáltalán szó), vagy a kártyák le vannak tiltva, akkor **false** jön válaszként. Tehát csak egy dologban lehetsz biztos: ha a **GetIsNetworkAvailable** eredménye **false**, akkor nincs elérhető hálózat.

Megjegyzés: a **true** értéket adó válasznak elég nagy esélye van. Főleg ha számításba vesszük azt is, hogy a virtuális hálózati kártyáknak — mint pl. a *loopback adapter* vagy a virtuális gépeket futtató környezetekhez a hoszt gépen telepített szoftveres hálókártyák — mindig van kapcsolatuk.

Az előbb írt metódust meg is kell hívni. Praktikus már az alkalmazás inicializálásakor — például a **MainPage** osztály konstruktorában megtenni ezt:

```
public MainPage()
{
    InitializeComponent();
    store = (ProductStore)DataContext;

    UpdateNetworkStatus();
}
```

Szüntesd meg a gépeden az összes hálózati kapcsolatot, és indítsd el F5-tel az alkalmazást! Ha minden jól ment, piros háttérrel kell, hogy láss.

Ha most visszaállítod a hálózati kapcsolataidat, az alkalmazás továbbra is piros háttérű marad, hiszen csak indításkor ellenőrizted a hálózat elérhetőségét. Itt az ideje a hálózat folyamatos figyelésének! Továbbra is a konstruktorba, az előbb beírt **UpdateNetworkStatus**-t hívó sor alá írd egy újabb sor kódot, amivel feliratkozol a fentebb említett **NetworkAddressChanged** eseményre:

```
NetworkChange.NetworkAddressChanged += NetworkChange_NetworkAddressChanged;
```

Ehhez a += után szereplő metódust is meg kell írni. Ennek törzse csak a már használt **UpdateNetworkStatus** metódust hívja, így biztosíthatod, hogy futásidőben is értesüljön programod a hálózati változásokról.

```
void NetworkChange_NetworkAddressChanged(object sender, EventArgs e)
{
    UpdateNetworkStatus();
}
```

Mindez persze nem elég ahhoz, hogy megbizonyosodj a távoli erőforrás elérhetőségéről, de nagyon gyorsan megtudhatjuk, hogy érdemes-e egyáltalán keresni azt. Sajnos, a Silverlightben hiányolnunk kell a .NET hasonló nevű névterében megtalálható hasznos **Ping** osztályt, amivel pontosabban tudnánk ellenőrizni célpontunk elérhetőségét. Így csak azt teheted, hogy megpróbálsz a kommunikációt — amennyiben a **NetworkInterface** osztály zöld utat adott —, de egyúttal hibakezeléssel fel is kell készülnöd az esetleges sikertelenségre.

Out-of-Browser applications

A Silverlight alkalmazás egy kliensoldali alkalmazás. Azért a böngészőben nézzük, mert ez az Internet természetes otthona. A böngészőtől azonban meg is tudunk szabadulni, és ezután az alkalmazásunk már úgy néz ki, mintha egy szokásos desktop alkalmazás lenne. Kényelmesebb elindítani, jobban illeszkedik a többi alkalmazásunk közé. Akár még több jogosultságot is kaphat, mint amivel egy böngészőben futó alkalmazás rendelkezik, megtartva a biztonságos működést, és mégis használhatóbbá téve a programot.

Kitörés

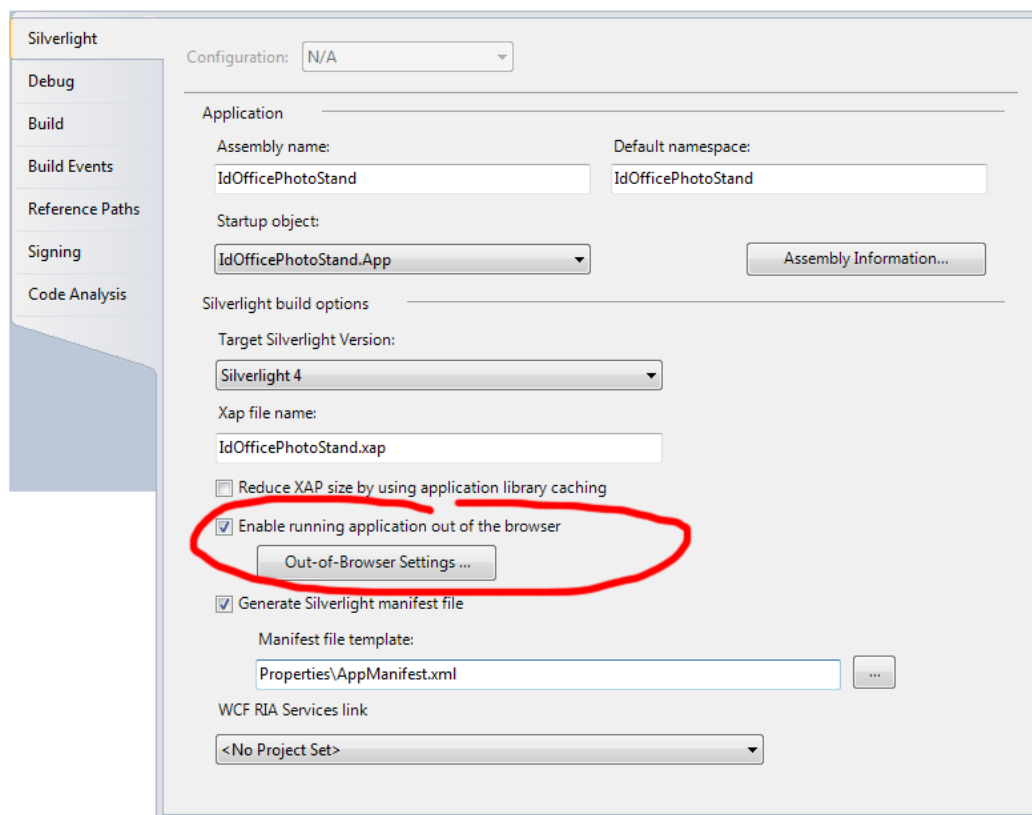
Hogy megszabadulj a böngésző „börtönétől”, első lépésként csak egy megfelelő opciót kell a Visual Studióban beállítanod. Ez a hely a projekt tulajdonságainak lapján található a Silverlight fülön, és az „Enable running application out of the browser” feliratot viseli (9-8 ábra). A böngészőn kívül futó Silverlight alkalmazásokra gyakran csak OOB alkalmazásként találsz hivatkozást, ez az Out-Of-Browser kifejezésből jön.

Indítsd el az alkalmazást az F5 gombbal! Az alkalmazás a böngésző nélkül (Out-Of-Browser) indul el. Ez a környezet egy kicsit más, mint a böngészőn belüli. Éppen ezért nem mindig szeretnénk, hogy így induljon nyomkövetés (debug) közben.

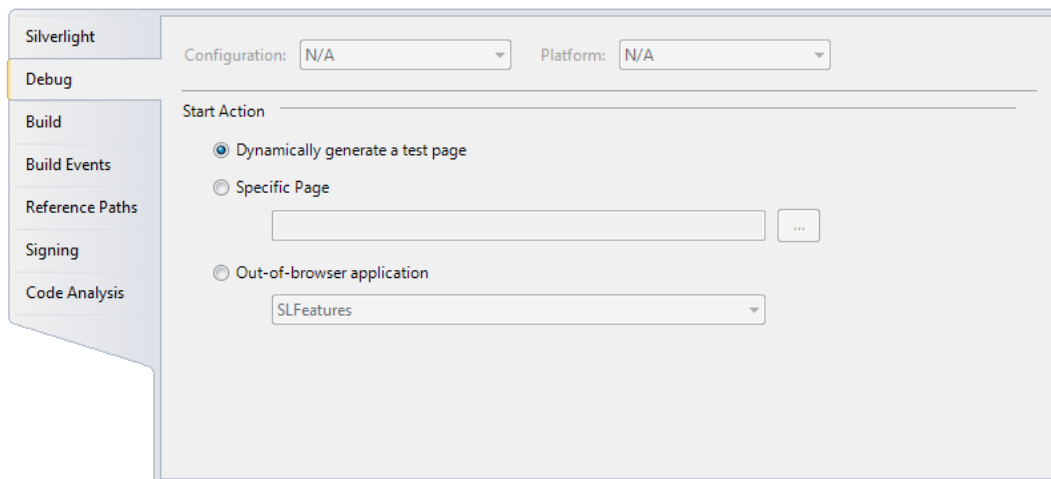
9. A Silverlight rejtett képességei

Egy Silverlight alkalmazás önmagában nem életképes, szüksége van egy futtató környezetre – hosztra. Ez alapvetően a böngésző, illetve a böngészőben futó Silverlight plug-in. Ha elhagyjuk a böngészőt, szükség van egy másik hosztra, amely saját ablakkal rendelkezik. Az OOB alkalmazások egy ilyen hosztban indulnak el.

Zárd be az alkalmazást! A Visual Studióban még mindig a Silverlight beállításait látod. Kattints a Silverlight alatti Debug fülre! Itt három opció közül lehet választani, amint a 9-9 ábra mutatja. Az első két opció valamelyikének választása esetében böngészőn belül követhetjük nyomon az alkalmazás működését. A harmadik opció esetén azonban böngésző nélkül végezhetjük a hibakeresést. Ezek a beállítások csak a fejlesztés közben jutnak érvényre!



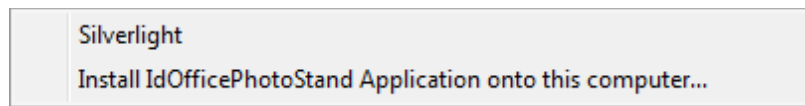
9-8 ábra: A Silverlight projekt tulajdonságlapja



9-9 ábra: A Debug fül

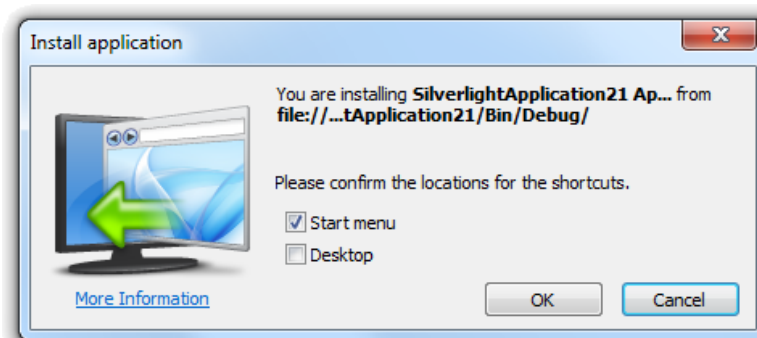
Hogy megnézhessd egy ilyen alkalmazás futásidejű viselkedését, válaszd ki az első, a „Dynamically generate a test page” feliratú opciót, és indítsd el az alkalmazást az F5 gombbal!

Az alkalmazás újra egy böngészőben látható, de most, ha azon bárhol jobb-kattintasz, megjelenik egy új menüpont, az „Install xy Application onto this computer...” (9-10 ábra). Próbáld is ki!



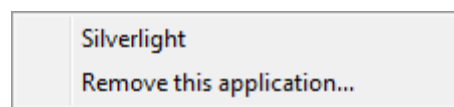
9-10 ábra: Silverlight context menü, telepíthető OOB alkalmazás esetén

A parancs egy telepítő dialógust indít el (9-11 ábra). Ebben a felhasználó kiválaszthatja, hogy hova szeretne ikont elhelyezni (asztal, start menü). Az OK gomb megnyomása után a szerveren tárolt Silverlight alkalmazás fájl letöltődik, lemásolódik a felhasználó gépére. Ebből a lokális cache-ből fogja majd elindítani az OOB hoszt. Létrejönnek a választott ikonok is. Az ikonok tulajdonságlapján megfigyelhetjük, hogy az SLLauncher.exe hoszt futtatja a letöltött SL alkalmazásokat.



9-11 ábra: A telepítő dialógus

Nyomd meg az OK gombot! Megjelenik az alkalmazás második példánya, de már nem egy böngészőben, hanem egy szokványos desktop ablakban. Ellenőrizd a Start menüben megjelenő ikont! Most bármelyik példányon jobb-kattintásra a telepítő menüpont helyén a „Remove this application...” menüpont szerepel (9-12 ábra). Kattints erre, majd zárd be a maradék példányt is!



9-12 ábra: A Silverlight context menüje telepített OOB alkalmazás esetén

Testreszabás

Elértük, hogy az alkalmazás futtatásához ne legyen szükség böngészőre, és azt egy ikonra kattintva indíthassuk. Sőt, futás közben sem egy böngésző belsejében, hanem saját ablakban látható.

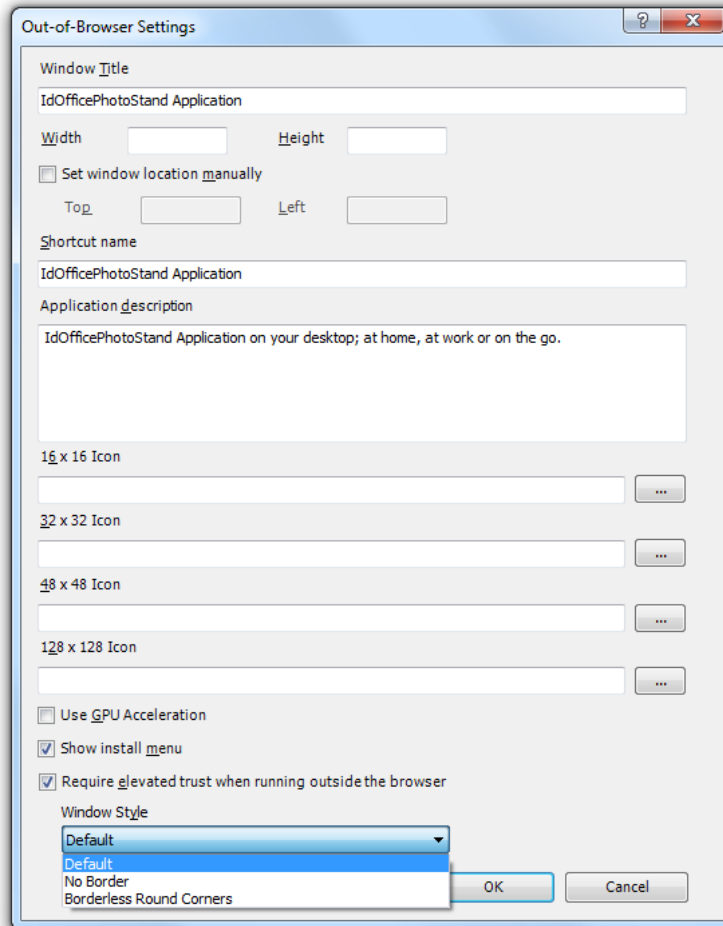
Ezt a saját ablakot és a telepítéskor megjelenő dialógust bizonyos keretek közt testre szabhatod.

Kattints vissza a projekt tulajdonságlapjának Silverlight fülére! Van ott egy „Out-of-browser Settings...” feliratú gomb. Ez a gomb a megoldás kulcsa. Rákattintva a 9-13 ábrán látható dialógus jelenik meg.

Vegyük sorra a megjelenő dialógus tartalmát fentről lefelé! Először a böngészőt helyettesítő ablak fejlécének feliratát adhatod meg (Window Title). Alatta beállíthatod, hogy ez az ablak mekkora legyen (Width, Height) és hol (Top, Left) jelenjen meg. Ha ezek üresek, akkor a projekt fő ablakának leírásában, a **MainPage.xaml** fájlban megadott méretben fog megjelenni, és az operációs rendszer dönti el, hogy hol.

A „Shortcut name” az a szöveg, ami a telepítést és eltávolítást intéző menüpontban jelenik meg az alkalmazás nevéként, és a telepítés után megjelenő ikonok alá is ez a felirat kerül. Az „Application

descripton” mezőbe írtak tooltipként jelennek meg az ikon felett. Az ikon különböző méretű képeit PNG formátumban és a megadott méretekben (16, 32, 48 és 128 pixel) lehet elkészíteni. Az elkészült PNG fájlokat először be kell rakni a projektbe — például egy alkönyvtárba — ezután ebben a dialógusban ki tudod azokat választani. A 16 pixeles kép meg fog jelenni az alkalmazás ablakának fejlécén. A Start menüben és az asztalon megjelenő ikonok a megjelenítési beállításoktól függően a 16, 32 vagy 48 pixeles képeket használják. A 128 pixeles kép a telepítő-dialógusban jelenik meg.



9-13 ábra: OOB beállítások

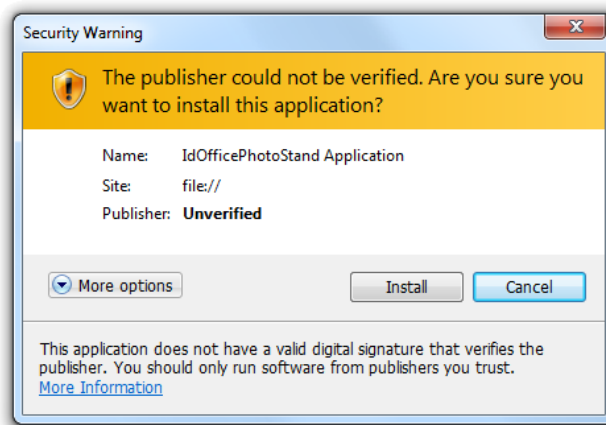
A kliensoldali hardveres grafikus gyorsítás kihasználására utasít a „Use GPU Acceleration” opció. Ezt csak akkor érdemes megjelölni, ha valóban intenzíven használunk grafikus effektusokat, animációkat.

A „Show install menu” opció törlésével eltüntetheted azt a menüt, amit korábban a telepítéshez használtunk az alkalmazáson jobb-kattintva. Most hagyd ezt megjelölve! Ez nem azonos azzal, hogy nem jelölöd meg az „Enable running application out of browser” opciót! Bár látszólag ez is megakadályozza a felhasználót a telepítésben, de ezt kódból helyettesíteni tudod.

A „Require elevated trust when running outside the browser” opció plusz jogosultságokkal futtatja az OOB alkalmazást. Ez bizonyos funkciók kihasználásához szükséges követelmény, mint például a „Window Style” legördülő listából való választáshoz, amely segítségével a kerettől akár teljesen meg is szabadulhatsz. Jelöld meg ezt az opciót, de a Window Style listát még ne állítsd át! Zárd be a dialógust, majd indítsd el az alkalmazást az F5 segítségével! Az továbbra is egy böngészőben jelenik meg. Jobb-kattintással próbáld telepíteni! Egy biztonsági figyelmeztetést kapsz, amint a 9-14 ábra mutatja.

A kiválasztott „elevated trust” opciónak köszönhető ez a figyelmeztetés. Az ezzel az opcióval telepített Silverlight alkalmazás ugyanis szélesebb jogkörökkel lesz elindítva a felhasználó gépén, mint az opció nélkül telepített. Ennek a bővített jogosultsághalmaznak a birtokában például el tudja indítani az Office alkalmazásokat, azokkal kommunikálni, műveleteket végeztetni tud. Mivel így akár „gonosz” kódot is

futtathatnának a felhasználó gépén, jobb, ha a felhasználó dönti el, hogy egy adott alkalmazást beenged-e ilyen esetben is, megbízik-e az alkalmazásban, illetve annak gyártójában.



9-14 ábra: Elevated Trust figyelmeztetés

Persze, ez az „elriasztó dialógus” megnehezíti alkalmazásunk népszerűsítését — sokan esetleg pont az üzenet miatt nem telepítik —, de ha figyelmesen elolvasod a szövegét, akkor kiderül, hogy ezt barátságosabbá lehet tenni — érvényes digitális kódalíró tanúsítvány segítségével.

API

Az eddig megismert lehetőségek, bár nagyon egyszerűen használhatók, néhány problémára (például a felhasználói felületen megjelenített telepítési lehetőség, automatikus programfrissítés, az ablak méretének és pozíciójának dinamikus beállítása vagy épp futásidőben a futtatás módjával kapcsolatos információk megszerzése) nem adnak megoldást.

Nézzük először a telepítést!

A telepítés egy Silverlight alkalmazás esetében azt jelenti, hogy az alkalmazás fájljait a telepítő eszköz lementi a felhasználó profiljába. A lementett programot az **sllauncher.exe** segítségével lehet elindítani. Az ikonok tulajdonságlapján is látni lehet, hogy azok valójában az **sllauncher.exe** fájlra mutatnak, persze megfelelően paraméterezve. Ahhoz, hogy a telepítés végbemenjen, mindenképpen a felhasználónak kell azt kezdeményeznie. Ennek egyik módja a Silverlight alkalmazás jobb-kattintásos menüjének használata. Azt is láttuk már, hogy ezt a menüt el is lehet tüntetni. Sokat nem veszít vele a felhasználó, hiszen úgysem találná meg (csak a gyakorlott felhasználók, akiket viszont fejlesztőnek hívnak). Sokkal praktikusabb ezt a lehetőséget a grafikus felületen meghirdetni. Ehhez csak el kell helyezni pl. egy gombot a felületen, aminek megnyomása végrehajtja a kódot. A telepítéshez szükséges kód egy sorban leírható:

```
Application.Current.Install();
```

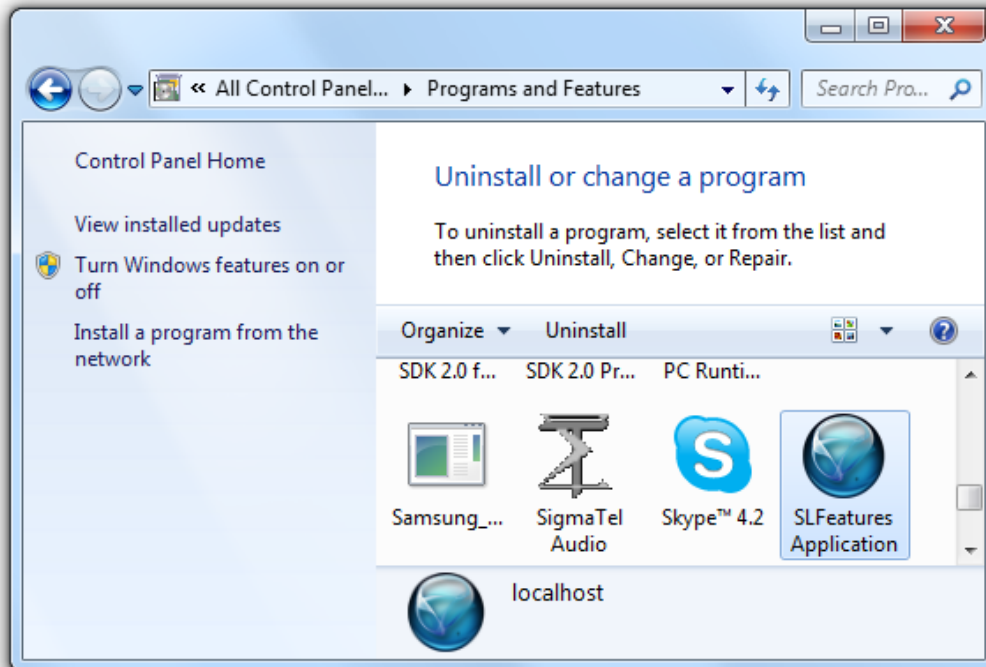
Ezt a metódust egy már telepített alkalmazásban végrehajtva egy kivételt kapsz, ezért hasznos előre ellenőrizni, hogy már telepítve van-e az alkalmazás. Egy telepített Silverlight alkalmazás az eredeti URL-en a böngészőben is elérhető, de attól az még telepítve van. Ezt az állapotot az **InstallState** felsorolt tulajdonság jelzi az alkalmazás egy példánya kapcsán. Tehát, az egyszerű sor helyett inkább használd az alábbi vizsgálatot:

```
if (Application.Current.InstallState == InstallState.NotInstalled)
    Application.Current.Install();
```

Az **InstallState** állapotától a felhasználói felületen lévő gomb láthatóságát, engedélyezettségét is függővé teheted. Ebben segíthet az, hogy a tulajdonság értékének változásairól is értesülhetsz az **InstallStateChanged** eseményen keresztül:

```
Application.Current.InstallStateChanged += new EventHandler(Current_InstallStateChanged);
```

A telepítés és kipróbálás után a felhasználó gyakran szeretné a programot eltávolítani a gépről. De az **Application** objektum **Install** metódusának nincs **Uninstall** párja! Ezt a feladatot továbbra is a jobb-kattintásos menüből tudja elvégezni. Ha ez nem szimpatikus, a telepített Silverlight alkalmazás megtalálható a szokásos helyen az operációs rendszer telepített programjainak listájában is, amint azt a 9-15 ábra mutatja.



9-15 ábra: A telepített Silverlight alkalmazás

A telepítettséghez hasonlóan vannak egyéb olyan állapotok is, amikről érdemes futásidőben tudomást szerezni. Így például megállapíthatod, hogy az elindított példány a böngészőben vagy azon kívül fut-e:

```
if (Application.Current.IsRunningOutOfBrowser) { ... }
```

Ellenőrizheted, hogy vajon az extra jogok birtokában fut-e a kódod:

```
if (Application.Current.HasElevatedPermissions) { ... }
```

A telepített alkalmazás, mint korábban említettem, a felhasználó profiljában van elmentve. Tehát ha a szerveren frissítik az alkalmazást tartalmazó XAP fájlt, arról mit sem tud a felhasználó gépe, ahol az ikon a régebben elmentett változatra mutat. Ezt a lokálisan elmentett változatot kellene felülírni ahhoz, hogy a friss változat indulhasson el. Ebben segít az **Application** objektum **CheckAndDownloadUpdateAsync** metódusa. Ez a metódus visszanéz a szerverre és ellenőrzi, hogy az ott tárolt XAP fájl újabb-e, mint a lokálisan tárolt. Ez persze hálózati kommunikációval jár, és mint minden hálózati kommunikáció a Silverlightban, ez is aszinkron hívás. Az aszinkron hívásokhoz egy esemény is tartozik, amely az eredmény megérkezésekor kerül jelzésre. Az automatikus frissítést praktikusán az alkalmazásunk indításakor érdemes elvégezni.

Az **App.xaml.cs** kódban először a konstruktorban iratkozz fel a **Startup** eseményre:

```
public App()
{
    this.Startup += this.Application_Startup;
    //egyéb konstruktori teendők...
    InitializeComponent();
}
```

A **Startup** eseménykezelőben feliratkozhat a **CheckAndDownloadUpdateCompleted** eseményre, majd meghívhatod a friss változatot ellenőrző és letöltő metódust:

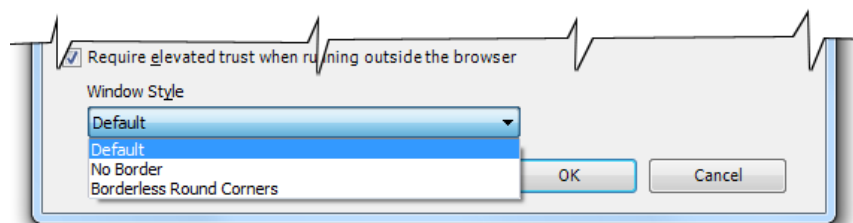
```
private void Application_Startup(object sender, StartupEventArgs e)
{
    this.CheckAndDownloadUpdateCompleted += CheckAppForUpdates;
    this.CheckAndDownloadUpdateAsync();
}
```

A frissítést jelző eseménykezelőben pedig ellenőrizheted, hogy van-e újabb alkalmazás-változat. Ha igen, akkor azt a **CheckAndDownloadUpdateAsync** metódus már le is töltötte. Ekkor az a már futó alkalmazás alatt cserélte ki a fájlt, tehát az új változat használatához újra kell indítani a programot. Sajnos ezt nem teheted meg kódból, ezért csak egy üzenetet dobhatsz fel, amelyben megkérheted a felhasználót az alkalmazás újraindítására.

```
void CheckAppForUpdates(object sender,
    System.Windows.CheckAndDownloadUpdateCompletedEventArgs e)
{
    if (e.UpdateAvailable)
        MessageBox.Show("Az alkalmazás frissült, kérem indítsa újra.",
            "Automatikus frissítés",
            MessageBoxButton.OK);
}
```

Ennek az automatikus „önfrissítésnek” van egy alapvető szépséghibája: kizárólag tanúsítvánnyal rendelkező alkalmazással működik!

Az OOB alkalmazás megjelenés szempontjából legfontosabb eleme az, hogy „levetkőztük” a böngésző keretet. Az alkalmazás még tovább vetkőztethető! Az Out-of-browser Settings dialógus alján, ha az elevated trust opció be van jelölve, három keretstílus közül választhatsz, amint azt a 9-16 ábra mutatja.



9-16 ábra: A keretmentes ablak beállítási lehetőségei

A Default a szokásos operációs rendszer által nyújtott keretet adja. A másik kettő esetében viszont még attól is megszabadulhatunk. Az utolsót választva kellemesen lekerekített sarkokkal találkozhatunk. A keret megszüntetésével azonban elveszíted a fejléctet is! Pedig az ablakunk mozgatásához a keretre is szükség van! Elég frusztráló egy olyan alkalmazás, amelyik stabilan áll a felbukkanás helyén, se mozgatni, se átméretezni, de még kilépni sem lehet belőle a szokásos keret hiányában.

9. A Silverlight rejtett képességei

A mozgathoz szemeljünk ki egy megfelelő vizuális elemet az ablakodon! Legyen ez az egész ablakot reprezentáló **UserControl** objektum, iratkozz fel annak **MouseLeftButtonDown** eseményére:

```
<UserControl ... MouseLeftButtonDown="UserControl_MouseLeftButtonDown">
```

Az eseménykezelőben — feltéve, hogy nem a böngészőben fut a program — hívd meg az ablak **DragMove** metódusát!

```
private void UserControl_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    if (Application.Current.IsRunningOutOfBrowser)
        Application.Current.MainWindow.DragMove();
}
```

Voilà! A mozgathoz megoldva.

A fejléc nemcsak a mozgathoz miatt fontos, hanem ott helyezkedik el az alkalmazást bezáró gomb is. Egyszerűen tegyél oda egy gombot, és iratkozz fel a **Click** eseményére:

```
<Button Content="X" Click="Close_Click" />
```

Az eseménykezelőben aztán hívd meg az ablak **Close** metódusát:

```
private void Close_Click(object sender, RoutedEventArgs e)
{
    if (Application.Current.IsRunningOutOfBrowser)
        Application.Current.MainWindow.Close();
}
```

A lezárást biztosító gombhoz hasonlóan elhelyezhetünk az ablakot minimalizáló és maximalizáló gombokat is. Ezeknek a gomboknak a **Click** eseménykezelőjük sem bonyolultabb az előbb látottnál:

```
private void MinimizeButton_Click(object sender, RoutedEventArgs e)
{
    Application.Current.MainWindow.WindowState = WindowState.Minimized;
}

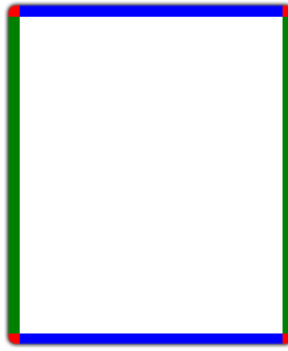
private void MaximizeButton_Click(object sender, RoutedEventArgs e)
{
    Application.Current.MainWindow.WindowState = WindowState.Maximized;
}
```

Újabb siker – a fejléc gombsorát is felügyeletünk alá vontuk!

Az eddigiekből látszik, hogy viszonylag szabad felügyeleti lehetőséggel rendelkezni az alkalmazás ablaka felett. Kódból még az ablak pozícióját és méretét is be tudod állítani:

```
Application.Current.MainWindow.Width = 300;
Application.Current.MainWindow.Left = 100;
```

Ezzel az ismerettel felvértezve akár az ablak átméretezését is megoldhatnád. Ne tedd! Pontosabban, ne így tedd! Az egérrel történő átméretezéshez ugyanis rendelkezésedre áll egy könnyen használható metódus, amit a **DragMove** mintájára **DragResize**-nak hívnak. Az alkalmazásablak átméretezéséhez hozz létre egy virtuális keretet, amint azt a 9-17 ábra mutatja!



9-17 ábra: Átméretező keret

Az összes irányba való átméretezéshez a **LayoutRoot** Grid végére helyezz el nyolc téglalapot! Ezek most színesek a szemléltetés miatt, de a végleges kódban minden **Fill** tulajdonság legyen **Transparent** értékre állítva:

```
<Rectangle Height="10" VerticalAlignment="Top" Fill="Blue"
    MouseLeftButtonDown="SizeTop"/>
<Rectangle Height="10" VerticalAlignment="Bottom" Fill="Blue"
    MouseLeftButtonDown="SizeBottom"/>
<Rectangle Width="10" HorizontalAlignment="Right" Fill="Green"
    MouseLeftButtonDown="SizeRight"/>
<Rectangle Width="10" HorizontalAlignment="Left" Fill="Green"
    MouseLeftButtonDown="SizeLeft"/>

<Rectangle Width="10" Height="10" HorizontalAlignment="Left" VerticalAlignment="Top"
    Fill="Red" MouseLeftButtonDown="SizeTopLeft"/>
<Rectangle Width="10" Height="10" HorizontalAlignment="Right" VerticalAlignment="Top"
    Fill="Red" MouseLeftButtonDown="SizeTopRight"/>
<Rectangle Width="10" Height="10" HorizontalAlignment="Left" VerticalAlignment="Bottom"
    Fill="Red" MouseLeftButtonDown="SizeBottomLeft"/>
<Rectangle Width="10" Height="10" HorizontalAlignment="Right" VerticalAlignment="Bottom"
    Fill="Red" MouseLeftButtonDown="SizeBottomRight"/>
```

Ez egy szépen keretezett ablakot ad. Figyeld meg, hogy mindegyik téglalapnak feliratkoztunk a **MouseLeftButtonDown** eseményére! Ezek implementációja hasonló minta alapján készült, itt a **SizeRight** metódus kódja látható:

```
private void SizeRight(object sender, MouseButtonEventArgs e)
{
    if (Application.Current.IsRunningOutOfBrowser)
        Application.Current.MainWindow.DragResize(WindowResizeEdge.Right);
}
```

A végére maradt még egy kis „finomság”. Helyezz el egy gombot például a felső gombsorban, és annak a **Click** eseményét kezelő metódusba írd be a következő sort:

```
Application.Current.MainWindow.TopMost = !Application.Current.MainWindow.TopMost;
```

Próbáld ki, érdemes!

Toast API

„Önnek új levele érkezett. Tocsi online”. Ilyen és hasonló üzenetekkel többnyire egy kicsi, magától megjelenő, majd egy idő múlva eltűnő ablakocskában találkozunk a képernyőnk jobb alsó sarkában. Bár ez a képesség nem a kiemelt fontosságúak közé tartozik, de bizonyos esetekben határozottan növelheti a szoftver használatának komfortérzetét. Az ilyen értesítőablakok megjelenítését a Silverlightban az ún. Toast API támogatja.

Hogy mi köze ennek a pirítóshoz? Talán csak annyi, hogy az ilyen ablakok általában a képernyő alján alulról felfelé szoktak beúszni, mint ahogy a pirítósz kiugrik a kenyérpíróból. Itt nagy hangsúly van a „képernyő” szón, arra utalva, hogy az animált értesítésnek az alkalmazás területén kívül kell megjelennie. Emiatt ez a lehetőség csak akkor használható, ha a Silverlight alkalmazásunk böngészőn kívül fut (OOB).

A „pirítósz” használatához annak teljes tartalmát nekünk kell megrajzolnunk. Szerencsére, ennek egyszerű módja van, akárcsak a nyomtatásnál.

Hozz létre a projektben egy új **UserControl** elemet **NetworkNotification** néven! Ez az objektum reprezentálja a figyelmeztető ablak tartalmát. Helyezz el itt egy **Border**-t és **abbanTextBlock** vezérlőt, ahogyan azt az alábbi kód is mutatja:

```
<Border Background="{Binding}" BorderBrush="Black" BorderThickness="2">
    <TextBlock Text="A hálózat állapota megváltozott"
        FontSize="14" FontWeight="Bold" Foreground="White"
        HorizontalAlignment="Center" VerticalAlignment="Center"/>
</Border>
```

Nyisd meg a **MainPage.xaml.cs**-t és keresd meg a korábban létrehozott **UpdateNetworkStatus** metódust, amely eredetileg ennek az alkalmazásnak a háttérszínét változtatta.

Első lépésként a metódus elé, közvetlen az osztály törzsében definiálj egy **NetworkNotification** típusú változót, és rögtön példányosítsd is! Közvetlen alatta definiálj egy **NotificationWindow** típusú változót, és példányosítsd:

```
NetworkNotification notification = new NetworkNotification();
NotificationWindow nw = new NotificationWindow();
```

A második pedig az ablak, ami az alkalmazástól függetlenül a teljes képernyő jobb alsó sarkában fog megjeleníteni. Ennek az ablaknak kell tartalomként megadni a **notification** változóban lévő **UserControl** objektumot. Írd át az **UpdateNetworkStatus** metódust az alábbi módon:

```
private void UpdateNetworkStatus()
{
    bool isNetwork = NetworkInterface.GetIsNetworkAvailable();
    if (!isNetwork)
        notification.DataContext = "Red";
    else
        notification.DataContext = "Green";

    nw.Content = notification;
    nw.Show(2000);
}
```

Látszik, hogy a **NotificationWindow** példányt annak **Show** metódusa jeleníti meg. Paraméterként átadható, hogy mennyi idő múlva tűnjön el (ezredmásodpercekben).

Ilyen ablakból egyszerre csak egyet lehet megjeleníteni, ezért is definiáltuk a **notification** változót osztályszinten. Több példány lehet ilyen ablakból az alkalmazásban, de akkor kellemetlen meglepetés érhet bennünket torlódó értesítések esetén. Amíg az egyik példány meg van jelenítve, a másik példányt nem lehet. Ha csak egyetlen példányunk van, ez nem fordulhat elő.

Sem a képernyő jobb alsó sarkában megjelenő ablaknak, sem pedig az abban elhelyezett **UserControl** objektumnak nem határoztad meg a méretét. A **UserControl** így akkora lesz amekkora az őt magában foglaló ablak, és ez így rendben is van. A megjelenő ablak mérete pedig 400*100 pixel. Ez az ő maximális mérete. Ha más méretet szeretnél, megjelenítés előtt beállíthatod a **NotificationWindow** példány **Width** és **Height** tulajdonságait, de ennél csak kisebbre.

A **Show** metódus párja a **Close**. Ezt akár a **NotificationWindow** kódjából is hívhatod (például **Click** eseményre, és így a felhasználó meg tudja gyorsítani a figyelmeztetés eltűnését).

Fájlkezelés

A drag & drop kapcsán megnéztünk egyfajta kommunikációt a külvilággal, az ott megismert fájlkezelés lehetőséget adott arra, hogy a merevlemezen tárolt fájlokat a felhasználó segítségével elérjük. Ha az alkalmazásnak adatokat is kell tárolnia, szükségünk lesz egy olyan tárhelyre, ahová biztonságosan és felügyelt módon menthetünk el adatokat. Ez az *Isolated Storage*.

Adatmentés és olvasás az *Isolated Storage*-ban

Az *Isolated Storage* gyakorlatilag egy (illetve több egyedi) könyvtár a felhasználó profiljában. Két szempont alapján jönnek létre ezek a könyvtárak: Minden Silverlight alkalmazásnak, illetve minden web site-nak saját kis terület jár. Így a könyvtár tartalmát vagy csak egy bizonyos alkalmazás, vagy csak egy bizonyos site-ról indított alkalmazások érik el. Ez szeparációt, tárterületek izolálását jelenti — innen ered a neve. Ki tilthatja meg nekünk, hogy bármelyik könyvtárba írjunk? Hát a futtatókörnyezet! S mivel a futtatókörnyezet erre figyel, több feladatot is megold egyszerre.

Az alkalmazás írójának tudnia kellene, hol is van ez a könyvtár. Nos ezt nem fogja megtudni sosem! Ez rögtön egyszerűbbé is teszi a szeparációs elv betartását. Konkrét fizikai útvonal és tárhely helyett egy API-t kapunk, amely segítségével nevesített stream-ekhez tudunk hozzáférni. Ezek a streamek írhatók is, és az API biztosítja, hogy azok mindenképpen a mi alkalmazásunkhoz és site-unkhhoz tartozó elkülönített könyvtárban helyezkedjenek el.

Ha egy alkalmazás írhat a merevlemezre akkor azt már senki sem állíthatja meg. Csak ír, ír és ír, egészen addig, amíg be nem telik a teljes partíció. Ezzel akár az operációs rendszer munkáját is lehetetlenné teheti. Ha ez előfordulhatna, sok problémát okozhatna a felhasználónak. Természetesen, a Silverlight erre is figyel! Az *Isolated Storage* tárterületeinek méretkorlátja van, amelyet nem léphet túl az alkalmazás. Persze minden megoldás újabb problémákat szül. Hogyan kezdünk valamit egy olyan alkalmazással, amelynek kevés az *Isolated Storage* által meghatározott tárterület? Úgy, hogy ezt *Isolated Storage* példányonként beállítható, a felhasználótól újabb terület kérhető!

Nézzük meg, hogyan működik ez a gyakorlatban!

Az alkalmazás fő ablakára (**MainPage**) húzz fel a Toolbox-ról két újabb gombot az eddigiek mellé! Az első nevezd el **MentesGomb**-nak, a másodikat pedig **BeolvasasGomb**-nak! Feliratozd őket az adott nevek alapján!

A mentés gombon dupla kattintással iratkozz fel a **Click** eseményre! Ez az eseménykezelő valósítja meg az adatok *Isolated Storage*-ba mentését. Ennek a tárhelynek a kezelését több Silverlight objektum is támogatja. Ezek közül az egyik leggyakrabban használt az **IsolatedStorageFile**, amelynek két statikus metódusa és egy statikus **IsEnabled** tulajdonsága van. Az *Isolated Storage*-ot le is tilthatja a felhasználó, ennek a tiltásnak a tényét olvashatjuk ki az **IsEnabled** tulajdonságból.

A két metódus pedig a már említett kétféle szeparáció (alkalmazás és web site) szerinti elkülönített tárterületekre ad vissza referenciát. Használd a **GetUserStoreForApplication** metódust az alábbi példa szerint:

```
IsolatedStorageFile file = IsolatedStorageFile.GetUserStoreForApplication();
```

A **file** példány azonban még nem maga a fájl, amibe írni lehet! A metódusai között található többek között az **OpenFile** és a **CreateFile** metódus, amelyek egy-egy streamet adnak vissza. Az így kapott stream segítségével lehet egy állományba adatot írni:

```
IsolatedStorageFileStream stream = file.CreateFile("adatok");
```

Az itt megadott név a virtuális fájl neve. Ez csak a név, fizikai útvonal nélkül, hiszen az Isolated Storage olyan mint egy elkülönített partíció, aminek most a gyökérkönyvtárban jársz. A **stream** példány már egy igazi stream, és a szokásos eljárásokkal írható is. Ebben a példában csak egy egyszerű szerializációt végzünk. Ehhez azonban szükséged lesz **System.Xml.Serialization.dll**-re, add a megfelelő referenciát hozzá a projekthez!

Az eseménykezelő metódus az eddigiek alapján tehát az alábbi kódot tartalmazza:

```
private void MentésGomb_Click(object sender, RoutedEventArgs e)
{
    using(IsolatedStorageFile file = IsolatedStorageFile.GetUserStoreForApplication())
    using (IsolatedStorageFileStream stream = file.CreateFile("adatok"))
    {
        XmlSerializer ser = new XmlSerializer(typeof(ObservableCollection<Product>));
        ser.Serialize(stream, store.Products);
    }
}
```

Ezt már ki is próbálhatnád, de sok haszna nincs, amíg vissza nem tudod olvasni. Tehát próba előtt inkább írd meg a betöltés gomb eseménykezelőjét is! A tervezőnézetre váltva duplakattints a betöltés gombon, és a kapott eseménykezelőben az első sorba máris beírhatod ugyanazt a sort, ami az előző eseménykezelőben is az első sor volt! A többi sor azonban már a fájl létrehozása és írása helyett annak olvasását valósítja meg:

```
private void BetoltesGomb_Click(object sender, RoutedEventArgs e)
{
    using(IsolatedStorageFile file = IsolatedStorageFile.GetUserStoreForApplication())
    if (file.FileExists("adatok"))
    {
        using (var stream = file.OpenFile("adatok", FileMode.Open, FileAccess.Read))
        {
            XmlSerializer ser = new XmlSerializer(typeof(ObservableCollection<Product>));
            var p = ser.Deserialize(stream) as ObservableCollection<Product>;
            foreach (var item in p)
            {
                store.Products.Add(item);
            }
        }
    }
}
```

Ha a **store.Products.Add.**kezdetű sor elé beírsz egy **store.Products.Clear()**-t is, akkor az aktuális listát kicseréli, ha nem, akkor ahhoz hozzáfüzi azt, amit a fájlban talált.

Indítsd el az alkalmazást, az rögtön két termékkel indul! Nyomj egy mentés gombot, aztán egy betöltés gombot! Már négy termék van (hacsak nem írtad be a **Clear**-t).

Megjegyzés: Ez a kód nem menti a termékek képeit. Sőt ami még rosszabb, el is száll, ha van nekik, mivel a WriteableBitmap-ot az XmlSerializer nem bírja feldolgozni agyilag. Képek enkódolására itt most nem térek ki, de a hibát kikerülheted, ha a serializernek megmondod, a képpel ne foglalkozzon.

A Product.cs fájlban a Product osztály Picture tulajdonságának definíciója elé írd be egy XmlIgnore attribútumot!

```
[XmlIgnore]
public ImageSource Picture
{ ...
```

A feladattal készen is vagy, de egy dolgot még meg kell említenem. Ha a **file** nevű változó metódusait átnézted, feltűnhetett, hogy van könyvtárkezelés. Egy könyvtárba írást szemléltet a következő kódrészlet.

```
file.CreateDirectory("alkonyvtar");
var dumaStream = file.CreateFile(@"alkonyvtar\duma.txt");
StreamWriter writer = new StreamWriter(dumaStream);
writer.WriteLine("hello leo");
writer.Close();
dumaStream.Close();
```

A **CreateDirectory** akkor is lefut, ha már van ilyen az Isolated Storage-ban. Ez esetben azonban nem csinál semmit. Figyeld meg, hogy innentől a fájl neve egy relatív elérési út. Könyvtárból olvasáskor is ilyen relatív útvonalat használhatsz.

```
var dumaStream = file.OpenFile(@"alkonyvtar\duma.txt", FileMode.Open);
StreamReader reader = new StreamReader(dumaStream);
string duma = reader.ReadLine();
reader.Close();
dumaStream.Close();
```

Fájlnevek és könyvtárnevek listázására is van lehetőség a **GetFileNames** és a **GetDirectoryNames** metódusokkal. Vannak másoló és átmozgató metódusok is, és így teljessé vált a fájlkezelés az elkülönített kis homokozóban.

Beállítások mentése és olvasása Isolated Storage-ból

Ha az alkalmazásunk bizonyos futásidejű paramétereit meg szeretnénk jegyeztetni a következő indításig (pl. használt ablakméret), az eddig említett fájlkezeléssel megoldható ugyan, de fölöslegesen bonyolult. Az ilyen beállítás jellegű adatok mentésére használható az **IsolatedStorageSettings** osztály.

Keress meg a kódban a **MainPage** konstruktorát, és a konstruktor törzsének végére írd be az alábbi kódrészletet:

```
public MainPage()
{
    ...

    if (Application.Current.IsRunningOutOfBrowser)
    {
        if (IsolatedStorageSettings.ApplicationSettings.Contains("width"))
        {
            App.Current.MainWindow.Width =
                (double)IsolatedStorageSettings.ApplicationSettings["width"];
        }
        this.SizeChanged += new SizeChangedEventHandler(MainPage_SizeChanged);
    }
}
```

Az Isolated Storage miatt az OOB ellenőrzést végző feltételre nem lenne szükség, de most az ablakméretet manipulálok, aminek viszont csak böngészőn kívül futó alkalmazás esetén van értelme.

A következő feltételre azért van szükség, mert ha a megadott néven nem lett mentve semmilyen érték, annak olvasására irányuló próbálkozás kivételt eredményez. A két **if** biztonságos mélyén már el is éred az elmentett értéket az **ApplicationSettings** dictionary segítségével. Ezenkívül a már tárgyalt kétféle szeparáció másik fajtáját is használhatod a **SiteSettings** dictionary használatával.

A **SizeChanged** eseményre azért iratkoztál fel, hogy annak eseménykezelőjében az ablak szélességét elmenthesd.

```
void MainPage_SizeChanged(object sender, SizeChangedEventArgs e)
{
    IsolatedStorageSettings.ApplicationSettings["width"]=App.Current.MainWindow.Width;
}
```

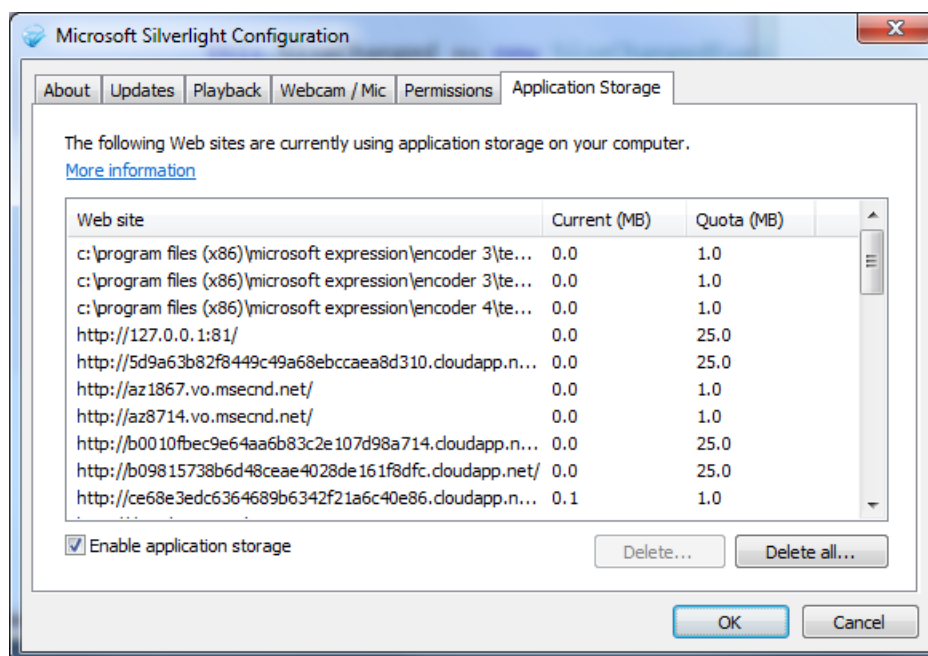
Megjegyzendő, hogy ettől még valójában nem menti az adatokat. Megjegyzi a memóriában. A merevlemezre csak az alkalmazás bezárásakor íródik ki. Ha ez valamiért nem elfogadható, akkor a Save metódus azonnali mentést biztosít.

```
IsolatedStorageSettings.ApplicationSettings.Save();
```

Ahogy nem menti magától azonnal, olvasni sem olvassa, csak az alkalmazás indításakor. Az olyan szituációkban, amikor két alkalmazás is fut egyszerre a kliens gépén és mindegyik használja a beállításokat, nincs mód arra, hogy futás közben az egyik alkalmazás beolvassa azokat, amelyeket a másik épp akkor mentett. Ehhez újra kell indítani azt. Szerencsére az ilyen szituációk igen ritkák.

Isolated Storage Quota

Az Isolated Storage-ba csak egy előre meghatározott mennyiségű adatot lehet maximálisan beírni. Hogy ez mennyi adatot is jelent, az több dologtól is függ. Egy böngészőben futó alkalmazás alapértelmezetten 1Mbyte-ot kap, a böngészőn kívül futó teljes jogú alkalmazás pedig 20Mbyte-ot. Ez jelentős különbség! Ezeket a beállításokat a felhasználó bármelyik Silverlight alkalmazás jobbkatintásos menüjében a Silverlight menüponttal érheti el. A Silverlight Configuration ablakban az Application Storage fül tartalmazza ezeket a beállításokat, amint azt a 9-18 ábra mutatja. Az „Enable Isolated Storage” opcióval egész Isolated Storage engedélyezhető vagy tiltható, illetve a Delete és Delete All gombokkal azok tartalma üríthető.



9-18 ábra: A Silverlight plug-in beállításai

Méretet nem lehet itt változtatni, azt majd az alkalmazás kéri, a felhasználó pedig jóváhagyja.

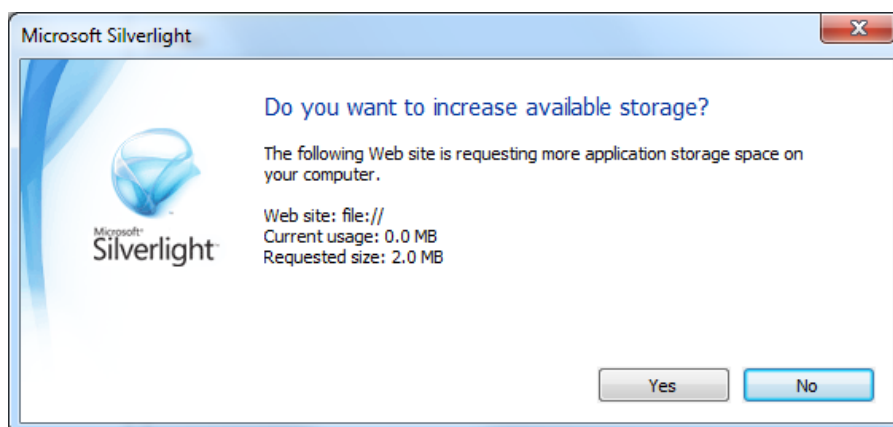
A rendelkezésre álló tárterület bármikor növelhető az **IncreaseQuotaTo** metódussal (egy **IsolatedStorageFile** példányon), ami a kívánt terület méretét byte-okban várja paraméterül. Ezt szemlélteti az alábbi kódrészlet:

```

if (file.AvailableFreeSpace < 1024)
{
    bool increased = file.IncreaseQuotaTo(file.Quota + file.UsedSize);
}

```

Az alkalmazás felelőssége a rendelkezésre álló hely ellenőrzése és biztosítása, szükség esetén a méret növelésének a kezdeményezése. Ebből azonban nem lehet kihagyni a felhasználót! Ezért az **IncreaseQuotaTo** metódus meghívása a 9-19 ábrán látható üzenetet eredményezi.



9-19 ábra: Az Isolated Storage méretének növelésekor megjelenő figyelmeztetés

Ha a felhasználó igennel felel, akkor megtörténik a tárterület növelés, egyéb esetben nem, és ezt a metódus eredményeként visszaadott **bool** érték közli is velünk.

Az igazi fájlrendszer elérése

Ebben a fejezetben már többször is a biztonsággal kapcsolatos elvárásokkal tevékenységekkel találkoztunk, szinte minden kisebb témában figyelembe kellett vennünk azokat. Elképzelhető ezek után, hogy a felhasználó beengedi az alkalmazást a féltve őrzött merevlemezére?

Igen! Azonban valahogy meg kell oldani, hogy a biztonsági követelmény ne csapjon át a használhatatlanságba. Ennek az a módja, hogy minden fájllelésről értesítést kap a felhasználó! Ez ebben a formában meglepőnek — akár ellenszenvesnek is — hangzik, de egy preventív figyelmeztetés egyszerűen megoldható, és összhangban van az elvárásokkal.

Az alkalmazás egyetlen fájlt sem érhet el, amíg a felhasználó fel nem hatalmazza annak elérésére, vagyis explicit módon meg nem mondja neki, hogy mely fájlt használhatja. A Silverlight ezt a megszokott fájlkezelő dialógusokkal kivitelezzi. Ezek a **SaveFileDialog** és **OpenFileDialog**. A jelentős különbség az, hogy ezek egyike sem fájlnevet ad vissza, hanem egy metódust biztosít, ami megnyitja olvasásra vagy írásra azt a fájlt, amit a felhasználó kiválasztott — bármi legyen is az. A kód csak egy **Stream** objektumot lát a folyamat eredményeként.

Ezek alapján a mentés gomb eseménykezelője Isolated Storage-ról igazi fájlrendszerre átírva a következőképpen néz ki:

```

SaveFileDialog save = new SaveFileDialog();
if (save.ShowDialog() == true)
{
    using (Stream stream = save.OpenFile())
    {
        XmlSerializer ser = new XmlSerializer(typeof(ObservableCollection<Product>));
        ser.Serialize(stream, store.Products);
    }
}

```

Az **OpenFileDialog** egy **FileInfo** példányt ad vissza. Ennek az objektumnak pedig több **OpenXY** metódusa, amivel megnyithatjuk a fájlt. Tehát a beolvasás így módosulhat:

```
OpenFileDialog open = new OpenFileDialog();
if (open.ShowDialog() == true)
{
    using (FileStream stream = open.File.OpenRead())
    {
        XmlSerializer ser = new XmlSerializer(typeof(ObservableCollection<Product>));
        var p = ser.Deserialize(stream) as ObservableCollection<Product>;
        foreach (var item in p)
        {
            store.Products.Add(item);
        }
    }
}
```

Rögtön feltűnik, hogy a betöltés esetén nem ellenőrzi a kód, hogy létezik-e az a fájl. Egyszerűen azért, mert itt bíztam a dialógusban, ami nem enged OK-t nyomni amíg nem egy létező fájlt választ a felhasználó. De ha fennáll a veszély, hogy az OK és a tényleges fájlnyitás közt történik valami azzal a fájljal, az **OpenFileDialog** egy **File** (illetve egyszerre több fájl kijelölése esetén **Files**) tulajdonsággal is rendelkezik, amiből kiolvashatók a kiválasztott fájl információi. Még a neve is! De vigyázat, az elérési út sosem!

Így akár mikor fájlt kell beolvasni vagy menteni, a felhasználó kap egy dialógust, amivel saját maga dönti el, hogy melyik legyen az a fájl.

My Documents

Képzeld el az előzőek alapján egy képkezelő alkalmazást, ami felsorolja a merevlemezen lévő képeidet, és megjeleníti azokat egy listában. Hát ilyen nem lehet megvalósítani, éppen ez a lényege a biztonságnak!

Azonban ilyen módon működő alkalmazásokra is nagy az igény, valamilyen értelmesen körülhatárolt megoldást tehát erre is biztosítani kell! Ilyen esetben a Silverlight kétpontos biztosítást használ. Egyfelől megköveteli a böngészőn kívüli futást, másrészt azon túl az elevated trust módra is szüksége van. Mindezek mellett nem engedi be az alkalmazást akármilyen könyvtárba!

Bizonyos könyvtárak elérési útját megkaphatod és azt szabadon használhatod. Ezek a könyvtárak az **Environment.SpecialFolder** felsorolásban találhatók. Ez azonban nemcsak a szabadon használhatókat, hanem azoknál jóval többet tartalmaz. A My Computer kivételével használható az összes **My** kezdetű könyvtár és a **Personal** is. A többi elérésének kísérlete egy **SecurityException**-nel ajándékoz meg. Az **SpecialFolder** felsorolásban nem az elérési út van, csak az ahhoz vezető kulcs. Az elérési utat az **Environment** osztály **GetFolderPath** metódusa adja vissza:

```
string mydocuments = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

Ezt a kiindulási útvonalat aztán összekombinálhatod fájlnevvvel vagy alkönyvtárakkal. A könyvtárból kifelé viszont nincs út. A megszerzett elérési úttal így néz ki a mentés kódja:

```
using (Stream stream = File.Create(System.IO.Path.Combine(mydocuments, "mentes.xml")))
{
    XmlSerializer ser = new XmlSerializer(typeof(ObservableCollection<Product>));
    ser.Serialize(stream, store.Products);
}
```

A fájl olvasása ezek után szinte magától értetődő:

```
string mydocuments = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
using (FileStream stream = File.OpenRead(System.IO.Path.Combine(mydocuments, "mentes.xml")))
{
    XmlSerializer ser = new XmlSerializer(typeof(ObservableCollection<Product>));
    var p = ser.Deserialize(stream) as ObservableCollection<Product>;
    foreach (var item in p)
    {
        store.Products.Add(item);
    }
}
```

Bár a fejlesztőnek nincs teljes szabadsága a fájlkezelés során, de az itt bemutatott lehetőségekkel gyakorlatilag minden valós probléma megoldható. Mindezek mellett a felhasználó is biztonságban érezheti magát.

Kommunikáció más alkalmazásokkal

A Silverlight otthonos kis védőburkot emel a fejlesztők köré. A kellemes fejlesztői környezet azonban még nem tud egyedül megküzdeni minden felhasználási helyzettel. Néha figyelünk, sőt vezérelnünk kell külső alkalmazásokat. A leggyakoribb ilyen feladat az Office alkalmazásokkal való kommunikáció. Ennek egyik módja, ha elindítjuk magát a vezérlendő alkalmazást. Ezt teszi lehetővé az **AutomationFactory** osztály.

Office (és más COM objektumok) vezérlése

Legfontosabb tudnivaló, hogy ez csak OOB Elevated Trust módban és csak Windowson működik. Ezt a kódban mindig ellenőrizni kell. Nemcsak Office alkalmazásokkal működik, hanem bármilyen COM komponenssel, de az Office a legkézenfekvőbb példa. Hogy COM objektumokkal kommunikálni tudjunk a felügyelt kódunkból, rengeteg apró, de fontos feladatot kell megoldania a futtató környezetnek. A legszembetűnőbb egy natív alkalmazással való kommunikációban, hogy a C# erősen típusos világához képest a processzorkódra lefordított bithalmazok nagyon távol helyezkednek el. Ezt egy huszárvágással oldották meg a Silverlight 4-es (ill. a .NET 4.0-ás) változatában. Bevezettek egy új adattípust, a **dynamic**-ot. Ez olyan típus, amely ún. *late binding* típusú műveletek elérését teszi lehetővé. Ez a COM világgal való kommunikáció megvalósításához szükséges kódot jelentősen lerövidíti. A late binding miatt nincs Intellisense támogatás, hiszen a program írása közben nem lehet előre tudni, hogy futásidőben mi is lesz az az objektum, amely majd egy **dynamic** típusú változóba belekerül, és így a műveletei sem lehetnek ismertek. A **dynamic** típus működése a **Microsoft.CSharp.dll**-ben van megvalósítva, ezt hozzá kell adni a projektünk referenciáihoz.

Helyezz el egy új gombot **MainPage**-en a gombsor végén! Nevezd el **ExcelGomb**-nak, és írd rá, hogy „Excelbe mentés”! A szokásos módon iratkozz fel a **Click** eseményére!

Az eseménykezelőben ellenőrizd le, hogy elérhető-e az **AutomationFactory**:

```
if (AutomationFactory.IsAvailable) { }
```

A kapcsos zárójelek közt indulhat a munka. Először a COM-tól kell szerezned egy Excel alkalmazást:

```
dynamic excel = AutomationFactory.CreateObject("Excel.Application");
```

Jelenítsd meg az Excel alkalmazás ablakát:

```
excel.Visible = true;
```

Az alkalmazás egyelőre üres. Egy **workbook**-ot kell hozzáadni, abban lesznek munkalapok, azokban pedig cellák:

9. A Silverlight rejtett képességei

```
dynamic workbook = excel.Workbooks.Add();  
dynamic sheet = workbook.ActiveSheet;
```

A cellákat sor és oszlop pozícióval lehet azonosítani, ezek 1-től kezdődően számozódnak. A fejléctet a következő kóddal lehet kitölteni:

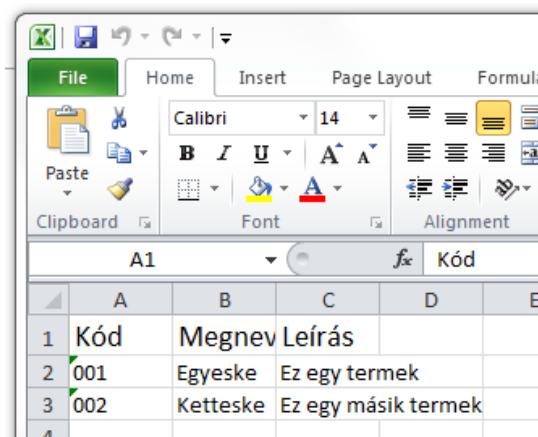
```
sheet.Range("A1:C1").Font.Size = 14;  
  
sheet.Cells(1, 1).Value = "Kód";  
sheet.Cells(1, 2).Value = "Megnevezés";  
sheet.Cells(1, 3).Value = "Leírás";
```

Az első sor mindhárom cellában megnöveli a betűméretet, aztán a cellákba egyenként szöveg kerül. Már csak az a dolgod, hogy egy ciklussal végiggyalogolj a terméklistán, majd minden egyes termék tulajdonságainak értékét egy-egy cellába beírd:

```
int pos = 2;  
foreach (var item in store.Products)  
{  
    sheet.Cells(pos, 1).Value = "'" + item.Code;  
    sheet.Cells(pos, 2).Value = item.Name;  
    sheet.Cells(pos, 3).Value = item.Description;  
    pos++;  
}
```

Az **item.Code** kiírásakor azért kell az aposztróf a karakter elé, mert az Excelnek ezzel lehet jelezni, hogy ez szöveg annak ellenére, hogy számnak látszik.

Az alkalmazás már futtatható, és az a 9-20 ábrához hasonló módon állítja elő a tartalmat.



	A1		
	A	B	C
1	Kód	Megnevezés	Leírás
2	001	Egyeske	Ez egy termék
3	002	Ketteske	Ez egy másik termék

9-20 ábra: Eredmény az Excelben

Kommunikáció Silverlight alkalmazások között

A Silverlight alkalmazások egymással is tudnak kommunikálni. Természetes, hogy ezt szerveren keresztül és hálózaton keresztül meg tudják tenni. Ha azok egy gépen találhatók és a kommunikációjuk szöveggént (egy stringben) leírható, akkor létezik egy ezeknél sokkal hatékonyabb módszer. A kommunikáció két résztvevőjét Sendernek illetve Receivernek hívják. A Sender küldi az első üzenetet. A Receiver fogadja azt, és esetleg válaszolhat rá. A két szerepkörhöz két osztály tartozik, a **LocalMessageSender** és a **LocalMessageReceiver (using System.Windows.Messaging)**, ezeknek az osztályoknak a példányai tudnak kommunikálni egymással. Ebből következően egy alkalmazás több irányban is kommunikálhat akár több — vegyes — szerepkörben.

Itt egy olyan példát készítünk el, amiben csak egy alkalmazás van, de ha annak több példánya van elindítva, akkor az egyik példány üzenetet küld a másiknak.

Bár a kommunikációt a Sender kezdi egy üzenet küldésével, de a küldés során meg kell nevezni a hozzá tartozó Receiver-t is. Ezért a Receiver felállításával kezd! Definiáld a szükséges osztályszintű változókat a MainPage-ben, és a konstruktorának elejére írd be az alábbi kódrészletet!

```
private LocalMessageReceiver messageReceiver;
private LocalMessageSender messageSender;

public MainPage()
{
    messageReceiver = new LocalMessageReceiver("SLFeatures");
    messageReceiver.MessageReceived += receiver_MessageReceived;

    try
    {
        messageReceiver.Listen();
    }
    catch (ListenFailedException)
    {
        //nem sikerült a hallgatóság,
        //feltehetően már fut egy példány ebből az alkalmazásból
    }
    ...
}
```

A **LocalMessageReceiver** létrehozásakor meg kell adnunk egy nevet. Ez a név gyakorlatilag a Receiver címe, ezért tehát egyedinek kell lennie. Az egyediség optimális esetben globális, de elég a tartomány szintű egyediség is. Hogy melyiket választod, azt a létrehozáskor egy második paraméterrel adhatod meg, ami egy kételemű felsorolás. Az alapértelmezett a tartomány szintű egyediség.

A létrehozott Receivernek fel kell iratkozni a **MessageReceived** eseményre, ha szeretnéd is megkapni a küldött üzeneteket. Ez az esemény akkor következik be, ha a Receiver elkezdett hallgatózni, és meghallott egy számára (tehát a létrehozásakor megadott névre) küldött üzenetet.

Fel kell készülni azonban arra, hogy ebből az alkalmazásból több is fut ugyanazon a gépen. Ilyenkor az előbb taglalt egyediség miatt csak az első számára sikeres a **Listen** metódus hívása. A további próbálkozások eredménye egy **ListenFailedException**. Ezért van a hallgatózás elkezdése egy **try-catch** blokkban.

Készítsd el a **receiver_MessageReceived** metódust! A küldött üzenetet az eseményparaméter **Message** tulajdonságában kapod meg, ez mindig egy string lesz. Ebben a példában két számot várhatsz benne — mert majd azt fogsz küldeni — szóközzel elválasztva, ezek az ablak méretét írják majd le. Értelmezd így a kapott stringet, és állítsd be az alkalmazás ablakának méretét!

```
void receiver_MessageReceived(object sender, MessageReceivedEventArgs e)
{
    string[] parts = e.Message.Split(' ');

    App.Current.MainWindow.Width = double.Parse(parts[0]);
    App.Current.MainWindow.Height = double.Parse(parts[1]);
}
```

Az üzenetet fogadó oldal már készen is van. Már csak a küldő oldal hiányzik. Most ugyanez az alkalmazás lesz az üzenetküldő is, ha nem tud fogadó lenni. A konstruktorban a **ListenFailedException** kivétel **catch** blokkjában hozz létre egy példányt a Senderből, és iratkozz fel a **SendCompleted** eseményre:

```
try
{
    messageReceiver.Listen();
}
catch (ListenFailedException)
{
    messageSender = new LocalMessageSender("SLFeatures");
    messageSender.SendCompleted += sender_SendCompleted;
}
```

A **SendCompleted** esemény akkor következik be, ha az aszinkron üzenetküldés végrehajtódott. Ebben az eseménykezelőben kaphatod meg az esetleges választ a másik alkalmazástól.

Választ úgy küldhetnél, ha az előbb megírt **MessageReceived** eseménykezelőben az esemény paraméterének kitöltenéd a **Response** tulajdonságát. Az így beállított értéket a Sender **SendCompleted** esemény paraméterének szintén **Response** nevű tulajdonságában éred el.

Az eddig leírt kóddal azt valósítottuk meg, hogy az első példány indulásánál az elkezd üzenetekre hallgatózni, az összes többi példány pedig előkészíti az üzenetküldést.

A **MainPage SizeChanged** eseménykezelőjében a Senderrel küldd el az aktuális ablakméretet:

```
void MainPage_SizeChanged(object sender, SizeChangedEventArgs e)
{
    if (messageSender != null)
        messageSender.SendAsync(string.Format("{0} {1}",
                                                App.Current.MainWindow.Width, App.Current.MainWindow.Height));
    ...
}
```

Indítsd el az alkalmazásodat kétszer a **Ctrl+F5** gombkombinációval! Próbáld átméretezni az ablakokat! Ha jól dolgoztál, a másodszorra indított alkalmazás átméretezése az először indított alkalmazás méretét is megváltoztatja.

Navigation Framework

Első kis alkalmazásaink békésen elférnek egy lapon. Aztán az egyre több funkció és az egyre bonyolultabb üzleti folyamatok lassanként alkalmazásképernyők tucatjait teszik szükségessé. Megtehetnénk, hogy minden képernyőt külön Silverlight alkalmazásban valósítunk meg, és azokat különböző HTML oldalakon helyezzük el, de ennél sokkal elegánsabb és több lehetőséget magában hordozó megoldás, ha ezeket a képernyőket egy alkalmazáson belül tudjuk kényelmesen váltogatni. A böngészőkben megszokott módon való navigálással, sőt akár a böngésző navigációs rendszerét kihasználva tehetjük ezt meg a Navigation Framework segítségével.

Két vezérlőt érdemes ehhez közelebbről megvizsgálni. Ezek egyike a **Frame**, amely képes arra, hogy saját belsejében megjelenítsen egy **UserControl** objektumot annak a projekten belüli útvonala alapján. Ezen túlmenően a **Frame** képes kommunikálni a böngészővel, tájékoztatva azt az éppen megjelenített információról, illetve lekérdezve, hogy mit is kell megjeleníteni. Ez több szempontból is nagyon hasznos.

Egyrészt az URL segítségével lehet vezérelni, hogy mi jelenjen meg a **Frame**-ben, másrészt ezeket az URL-eket a böngésző megjegyezheti. Később a navigáció során egy Silverlight alkalmazás belsejében lehet lépkedni. Ha pedig az URL határozza meg, hogy a Silverlight alkalmazás éppen melyik darabkáját láthatjuk, akkor azt az URL-t el lehet küldeni emailben, ki lehet rakni egy blogba, és így tovább. Tehát, egy URL nem csak az alkalmazás elejétől való indítását teszi lehetővé, hanem az alkalmazáson belüli jól meghatározott állapotokra navigálást is.

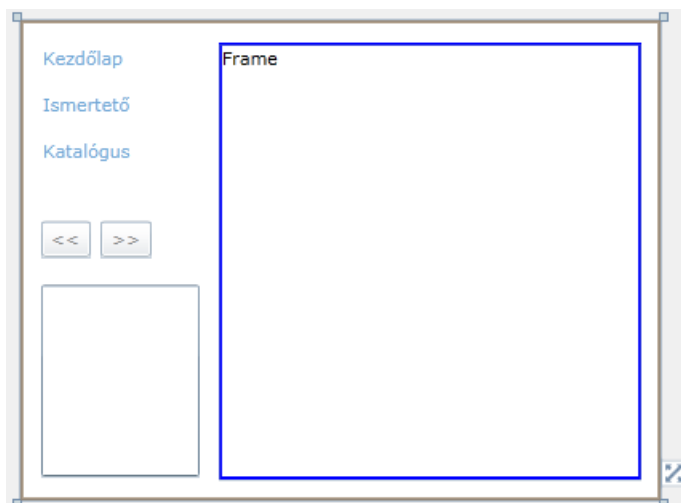
A webes fejlesztés során egyre fontosabb az ún. *Search Engine Optimization* (SEO). Ahhoz, hogy ez jól működhessen egy Silverlight alkalmazás esetében, fontos, hogy egyáltalán legyen olyan URL, ami a megfelelő ponton nyitja meg az alkalmazást.

A másik fontos vezérlő a **Page**, a **Frame**-hez hasonlóan ez is tudomást szerez a navigációs eseményekről. Hozzáfér az URL paraméterekhez, és navigációt is tud kezdeményezni. Bár nem kötelező, de praktikus a **Frame** vezérlőben való megjelenésre **Page** vezérlőket szoktunk használni.

Megvalósítás egyszerű vezérlőkkel

Hozz létre egy új Silverlight alkalmazást, anélkül, hogy hosztolnád azt egy webes alkalmazásban!

A **MainPage**-en helyezz el bal oldalon egymás alatt három **HyperlinkButton** vezérlőt/ Feliratozd őket a Kezdőlap, Ismertető és Katalógus szavakkal! Ezek alá helyezz egy új sorba két gombot egymás mellé << és >> jelekkel! Legalulra pedig tegyél egy **ListBox**-ot! A jobb oldalon fennmaradó szabad területre helyezz el egy **Frame**-et! A szemléletesség kedvéért adj neki színes körvonalat a **BorderBrush** és **BorderThickness** tulajdonságainak kitöltésével! A navigációs felületnek a 9-21 ábrán láthatóhoz hasonlóan kell lennie.



9-21 ábra: A navigációs példafelület

A projectben hozz létre egy új könyvtárat Views néven! Ebben a könyvtárban pedig hozz létre három új oldalt a Silverlight Page sablonnal! Ezek neve legyen **About.xaml**, **Catalog.xaml** és **Home.xaml**!

Ezeknek a fájloknak a XAML forrása a **<navigation:Page>** elemmel kezdődik. A C# kódban is tetten érhető az ős, mert ezek az objektumok a **Page** osztályból származnak. A **MainPage** a neve ellenére nem a **Page** osztály leszármazottja, hanem a **UserControl**-é. Ez a különbség nagyon fontos!

Mindhárom előbb létrehozott oldalra helyezz el egy-egy szövegblokkot megkülönböztető szöveggel! A **Page** vezérlő rendelkezik egy **Title** tulajdonsággal, ez a XAML fájlban megtalálható. Keresd meg ezeket a **Title** tulajdonságokat is, és írd át magyar megfelelőjükre! Az adott **Page** megjelenítésekor ez a **Title** tulajdonság meg fog jelenni a böngésző ablakának fejlécén.

A **MainPage**-en válaszd ki a Kezdőlap feliratú **HyperlinkButton** objektumot! Annak tulajdonságlapján találsz egy **NavigateUri** tulajdonságot. Ide kell beírni, hogy a gombra kattintva hova navigáljon az alkalmazás, ezt a projecten belüli relatív útvonallal adhatod meg. Ez ebben az esetben **/Views/Home.xaml** útvonal. Nem szabad megfélekedned a nyitó „/” jellel! Az Ismerető **NavigateUri** tulajdonságát állítsd **/Views/About.xaml**-ra!

Az utolsó gombbal most ne törődj, csak indítsd el az alkalmazást! Nem meglepő módon a **Frame** üresen jelenik meg. Kattints a Kezdőlap feliratú gombon (linken), és máris megjelenik a frame belsejében a **Home.xaml**.

Vizsgáld meg a böngésző fejlécét és címsorát! A fejlécen az a szöveg olvasható, amit a **Home.xaml** belsejében a **Page** objektum **Title** tulajdonságába írtál. A címsorban pedig a VisualStudio által automatikusan generált teszt HTML oldal elérési útja látható. Az URL végén a HTML fájl neve után azonban egy **#** és a gomb **NavigateUri** tulajdonságába beírt relatív útvonal látható. A **#** jel a HTML-nél egy oldalon belüli könyvjelzőre való navigálást jelenti. Ezt használja ki a Silverlight. Így nem kell elnavigálni az oldalról, tehát az alkalmazás nem veszti el az állapotát, adatait, nem töltődik újra az egész lap. A HTML

könyvjelző információját pedig felhasználja arra, hogy megállapítsa, melyik lapot kell a keretben megjeleníteni.

Írd át az URL végét **#/Views/Home.xaml**-ról **#/Views/About.xaml**-re! Megjelenik a frame belsejében az **About.xaml**. Így lehet egy URL segítségével az alkalmazás bizonyos részeit megcímezni (DeepLink).

Állítsd le az alkalmazást, és a **Catalog.xaml** üres területére húzz fel egy második **TextBlock** vezérlőt! Váltás át a lap C# kódjára! Ebben a kódban találsz egy **OnNavigatedTo** nevű metódust. Ez a metódus akkor fog meghívódni, amikor egy URL vagy egyéb navigáció ezt az oldalt kérte, ezért az megjelenítésre kerül. Töltsd ki az alábbi kódrészlet alapján:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    if (NavigationContext.QueryString.Contains("termek"))
        textBlock2.Text = NavigationContext.QueryString["termek"];
    else
        textBlock2.Text = "Nem lett termék megadva";
}
```

A **NavigationContext** hozzáférést biztosít az URL paraméterekhez. A kód első sora azt vizsgálja meg, hogy adtak-e át „termek” néven valamilyen adatot. Ha igen, akkor azt az adatot a második szövegblokkban megjeleníti, ha nem, akkor azt írja oda, hogy nem kapott adatot.

Indítsd el az alkalmazást és a böngésző címsorát egészítsd ki a következővel:

```
#/Views/Catalog.xaml?termek=Hatalmas monitor
```

Láthatod, hogy a megadott szöveg megjelenik a Silverlightban. Próbáld ki más értékekkel és a kérdőjeles rész nélkül is!

Apró szépséghiba, hogy az alkalmazás indításakor a **Frame** üresen jelenik meg. Ezt a **Source** tulajdonság kitöltésével tudod orvosolni.

A **Frame** osztály olyan metódusokkal is rendelkezik (**Navigate**, **CanGoBack**, **GoBack**, stb.), amelyek segítségével a lapok közötti navigációt programkódból is elvégezheted.

Lapozások közti adatmegőrzés

Vajon minden lapozáskor létrejön-e egy-egy új példány az adott lapból, vagy csak az első alkalommal, hogy aztán az mindig újra fel legyen használva?

Ha sok **Page** objektum van az alkalmazásban, akkor azok mindegyikéből egy-egy példány memóriában tárolása pazarlás lehet. Viszont ha a munkafolyamat megkívánja, vagy legalábbis engedélyezi az oda-vissza lapozgatást munka közben, jobb lenne megőrizni az előzőleg beírt adatokat.

Tegyél fel a **Home.xaml** lapra egy szövegdobozt, és indítsd el az alkalmazást! A kezdőlapra navigálva írd valamit a szövegdobozba, majd kattints az **Ismertető** gombra (linkre) és újra a kezdőlapra! Azt tapasztalod, hogy a szövegdoboz kiürült. Tehát minden lapozáskor egy vadonatúj példány jön létre a lapból. Hogy szövegdoboz ne ürüljön ki, többféle módon is megoldható:

Az első lehetőség, hogy kihasználva a **Page** osztály navigációs eseményeit (**OnNavigatedFrom**, **OnNavigatedTo**), a lapozáskor elmented illetve visszaállítod a szövegdoboz tartalmát. Ezeket az eseményeket a lap C# kódjában tudod használni.

A második módszer az, hogy ráveszed a Navigation Frameworköt, ne dobja ki a már egyszer létrehozott oldaladat. Nyisd meg a **Home.xaml**-t tervező nézetben! A **Page** tulajdonságlistáján találsz egy **NavigationCacheMode** tulajdonságot! Azt állítsd át **Required**-re. Ezzel utasítottad, hogy ha már egyszer létrehozta a lapot, többé ne felejtse el.

A harmadik módszer talán a legelegánsabb: használd ki az adatkontextust! Így nemcsak megőrzöd a lapozások közt az adatokat, de egyből meg is osztod az alkalmazás többi részével.

Adj a projecthez egy új osztályt **Adatok** néven! Készíts benne egy string típusú **Szoveg** nevű tulajdonságot!

```
public class Adatok: INotifyPropertyChanged
{
    public string Szoveg { get; set; }
}
```

A **MainPage** konstruktorában készíts eből az osztályból egy példányt, és helyezd el az adatkontextusban:

```
public MainPage()
{
    InitializeComponent();
    this.DataContext = new Adatok { Szoveg = "Helló adat" };
}
```

A következő fontos lépésként a **Home.xaml** tulajdonságainál a **NavigationCacheMode**-ot állítsd **Disabled** értékre! Enélkül az összes lap tulajdonsága a cache-ből kerülne kiolvasásra, tehát nem frissülne a másik lapon megváltoztatott adatokkal.

Megjegyzés: A cache kikapcsolása nem szükségszerű, de ha a cache-ből jön vissza az oldal, akkor már megtörtént rajta az adatkötés, amit most fogsz használni. Az adatkötés frissítéséhez az **Adatok** osztályon implementálni kellene az **INotifyPropertyChanged** interfészt. Ezt ebben a példában nem tesszük meg, de ha megtennéd, akkor a már felépített adatkötések is értesülnének az adat változásáról, így a cache-ben lévő oldal is.

Helyezz el egy **TextBlock** vezérlőt a **Home.xaml** lapon! A szövegét adatkötéssel kösd az adatkontextus **Szoveg** tulajdonságához!

```
<TextBlock Text="{Binding Szoveg}" Margin="20,100,0,0" Name="textBlock2"
    HorizontalAlignment="Left" VerticalAlignment="Top" />
```

Most helyezz el egy szövegdobozt az **About.xaml** lapon, és állíts be rá kétirányú adatkötést az alábbiak szerint:

```
<TextBox Text="{Binding Szoveg, Mode=TwoWay}" HorizontalAlignment="Left"
    Margin="20,100,0,0" Name="textBox1" VerticalAlignment="Top" Width="149" />
```

Indítsd el az alkalmazást, és nézd meg a kezdőlapon a szöveget! Navigálj át az ismertető lapra, és a szövegdobozban írd át a szöveget, majd nézd meg újra a kezdőlapon! Ott már az ismertetőlapon beírtakat kell látnod!

Egyéb lehetőségek

Vizsgáld meg újra az URL-eket! A „#” kulcsfontosságú, hiszen ebből tudja a böngésző, hogy nem kell újra letöltenie a lapot. Az utána következő rész azonban lehetne szebb is. Nem elegáns hogy mindig kell egy „/” jel legyen az elején. Az sem szép, hogy ott szerepel a projectben létrehozott könyvtár neve, a Views. Elnevezhettem volna magyarul is azt a könyvtárat, de egyrészt akkor is fölösleges, másrészt nem szerencsés keverni a kódoláshoz használatos elnevezéseket a felhasználó által ellátott elnevezésekkel. Ezt a problémát oldja meg az **UriMapper** osztály, amelynek segítségével relatív útvonalneveket képezhetsz le rövidebb kulcsokra.

A **Frame** osztály olyan metódusokkal is rendelkezik (**Navigate**, **CanGoBack**, **GoBack**, stb.), amelyek segítségével a lapok közötti navigációt programkódból is elvégezheted.

Egy ilyen **UriMapper** példányt praktikus alkalmazásszinten létrehozni. Nyisd meg az **App.xaml** fájlt!

Egy névtérhivatkozásra lesz szükséged. Az **Application** elem **xmlns** kezdetű attribútumai alá írd be a **System.Windows.Controls.Navigation** névtérhivatkozást az alábbi kódrészlet alapján.

```
<Application xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
              xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
              xmlns:nav="clr-
namespace:System.Windows.Navigation;assembly=System.Windows.Controls.Navigation"
              x:Class="NavDemo.App"
              >
```

Így most a XAML-ben deklaratíván létrehozatsz egy URI szótárt az **Application.Resources** blokk belsejében.

```
<Application.Resources>
  <nav:UriMapper x:Key="mapper">
    <nav:UriMapping Uri="Kezdolap" MappedUri="/Views/Home.xaml" />
    <nav:UriMapping Uri="Ismerteto" MappedUri="/Views/About.xaml" />
    <nav:UriMapping Uri="Katalogus" MappedUri="/Views/Catalog.xaml" />
    <nav:UriMapping Uri="Katalogus/{name}" MappedUri="/Views/Catalog.xaml?termek={name}"/>
  </nav:UriMapper>
</Application.Resources>
```

Az **UriMapper** elem **x:Key** attribútumának értéke segít majd erre az erőforrásra hivatkozni a **MainPage** kódjában.

Az **UriMapping** elemek mindegyike egy-egy **Uri** alias definíciót tartalmaz. Meglehetősen magukért beszélnek, de az utolsó felettébb érdekes. Az **Uri** értéke az, amit a böngésző címsorában a # után (vagy a linkek **NavigateUri** tulajdonságába, esetleg a frame **Source** tulajdonságába és mindenhova ahová egy urit kell írni) beírhat. Az utolsó sorban van ennek egy kapcsos zárójeles része, ami egy változó. A változó neve az, ami a kapcsos zárójelek közt van. Azt jelenti, hogy bármit írhat a # után ami **Katalogus/-rel** kezdődik. A / utáni rész a **name** nevű változóba kerül, és a **MappedUri** attribútumnál lesz felhasználva.

Nyisd meg a **MainPage.xaml**-t, és írd be a **Frame** elembe az **UriMapper** attribútumot!

```
<sdk:Frame UriMapper="{StaticResource mapper}" ...
```

Máris átírhatod a link gombok **NavigateUri** tulajdonságait. A **/Views/Home.xaml** helyett pl. egyszerűen írd azt, hogy **Kezdolap**! Az **Ismerteto** link **NavigateUri** tulajdonsága legyen egyszerűen **Ismerteto**, a **Katalógusé** pedig **Katalogus**!

Indítsd el az alkalmazást és próbáld ki az új linkeket! A címsorba kézzel is beírhatod őket. Figyeld meg azonban, hogy ettől a régi formátumú uri még elérhető, csak már a rövidített is használható! Próbáld ki a **#Katalogus/Ez a termék** URI végződést!

Silverlight Browser Integration

Gyakran szükség lehet arra, hogy egy meglévő weblapra apróbb kiegészítéseket kell készíteni. Ezt Silverlight segítségével is meg lehet tenni, ekkor a Silverlight kód a böngészőbe ágyazott plugin segítségével kommunikál a böngészővel. Ezt az eljárást *Silverlight Browser Integration*-nek (SBI) nevezzük. Az SBI megismeréséhez kis trükkel egy nagyon egyszerű tesztalkalmazást állíthatsz össze.

Hozz létre Visual Studióban egy új Silverlight Application projektet **SBIDemo** néven, hoszt web alkalmazás nélkül!

Indítsd el az F5 megnyomásával, majd állítsd is le! Ez a lépés arra jó, hogy a környezet létrehozzon egy olyan HTML oldalt, amibe már bele van ágyazva a Silverlight alkalmazás, ugyanis a feladat megoldásához pontosan erre a HTML oldalra lesz szükség.

A Solution Explorer fejlécén kattints rá a Show All Files ikonra! A megjelenő mappák közül nyisd meg a **bin\Debug**-ot!

A **Debug** mappában található HTML fájlt húzd rá a projekt nevére a fában! Közvetlen a **MainPage.xaml** alatt fog megjelenni. Jobb-kattints rajta, és a menüből válaszd az „Include In Project” parancsot! Nyisd meg ezt a lapot szerkesztésre, és módosítsd a

```
<param name="source" value="SBIDemo.xap"/>
```

sort az alábbira:

```
<param name="source" value="bin/Debug/SBIDemo.xap"/>
```

Mentsd el, majd válaszd ki a projekt alapértelmezett lapjaként (Set As Start Page)!

A Silverlight plug-in

Először is kezelni kell azt az esetet, ha az oldal olyan böngészőben fut, amely még nem telepítette a Silverlightot. Ezt az alábbi kódrészlettel teheted meg:

```
<object type="application/x-silverlight-2" data="data:application/x-silverlight-2,"
    width="100%" height="100%">
    <param name="source" value="ClientBin/MyApp.xap"/>
    <param name="minRuntimeVersion" value="2.0.31005.0" />
    <param name="autoUpgrade" value="true" />
    <param name="background" value="white" />
    <param name="onerror" value="onSilverlightError" />

    <a href="http://go.microsoft.com/fwlink/?LinkId=124807">
        
    </a>
</object>
```

Az **object** elem itt négy attribútummal rendelkezik. A **type** attribútum árulja el a böngészőnek, hogy erre a helyre Silverlight plug-int kell használni. A **data** attribútum néhány böngészőtípusnak kell — azonos tartalommal, mint a **type**.

A **width** és **height** attribútumok meghatározzák, hogy mekkora méretben jelenjen meg a Silverlight tartalom az oldalon. Meg lehet adni pixelben és százalékosan.

Az **object** elem belsejében a plug-innek szóló indítási paraméterek vannak felsorolva **param** elemekként. A paraméterek után pedig az a HTML részlet, amelynek akkor kell megjelennie, ha a böngésző nem ismeri a Silverlightot, ez ebben az esetben egy link a Silverlight telepítőjére.

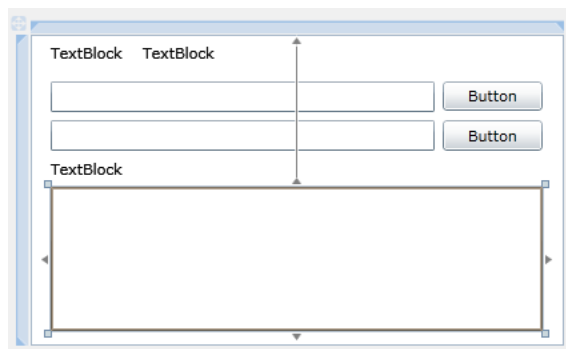
A paraméterekből a legfontosabbak láthatók a kódrészleten. A **source** paraméter kötelező, hiszen ez mutatja meg a plug-innek, honnan töltheti le a programot, azaz a XAP fájlt.

A **minRuntimeVersion** gyakorlatilag azt közli, hogy a letöltendő XAP fájl a Silverlight melyik verziójával lett elkészítve. Ebből tudhatja a böngésző azt, ha túl régi verziójú plug-in van telepítve. Ilyen esetben a következő paraméter, az **autoUpgrade** állapotától függően automatikusan telepítheti az újabb Silverlight változatot.

A **background** paraméterrel lehet beállítani, hogy milyen legyen a plug-in háttérszíne. Próbaképp írd át ennek az értékét **#ffff00**-ra (sárgára)! Nyomd meg az F5 gombot! Ha minden jól sikerült, akkor a böngészőben csak egy nagy fehérséget látsz, ami Silverlight alkalmazás.

Állítsd le az alkalmazást! Ha megnézed a XAML fájlt, láthatod, hogy a **LayoutGrid** háttérszíne fehérre van állítva. Töröld ki mindenestől a háttér beállítást erről a Grid-ről, ez a Silverlight esetében azt jelenti, hogy a háttér átlátszó.

Alakítsd ki a lapon a 9-22 ábrán látható felületet! Két **TextBlock** az első sorba, aztán egy **TextBox** és egy **Button** kétszer egymás alá, majd egy újabb **TextBlock** és a maradék helyre egy **StackPanel**.



9-22 ábra: SBI tesztfelület

Most indítsd el újra az F5-tel! Sárga háttéren jelenik meg a gomb. Azért van ez így, mert az alkalmazásnak átlátszó a háttere, mögötte viszont a Silverlight plug-in látszik, aminek a paraméterében a sárga színt határoztad meg háttérszínként (#ffff00 értékkel).

Beállíthatod a Silverlight plug-int úgy, hogy az ne töltsse ki az egész képernyőt:

```
#silverlightControlHost {  
    height: 50%;  
    text-align:center;  
}
```

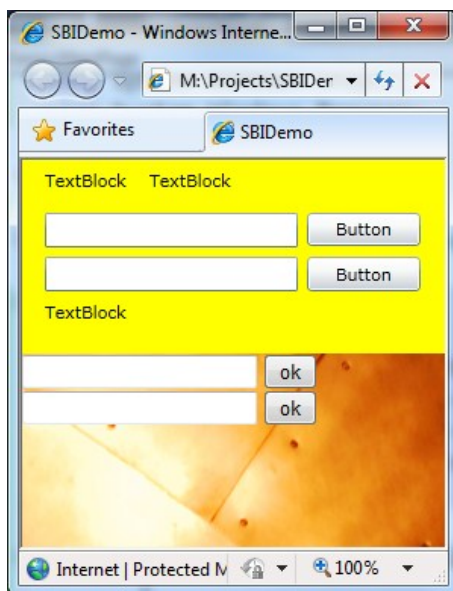
Hogy szemléletes legyen az átlátszatlanság illetve átlátszóság, adj háttérképet a HTML laphoz! Keress egy képet, és hegyszerűen húzd rá a fájlrendszerből a Visual Studio Solution Explorerében a projekt nevére! Módosítsd a **body** stílusát:

```
body {  
    padding: 0;  
    margin: 0;  
    background-image: url('img13.jpg');  
}
```

Helyezz el a **form** elembe néhány újabb HTML elemet:

```
<form>  
...  
    <input id="duma" type="text" /> <input id="gomb1" type="button" value="ok" />  
    <br />  
    <input id="parameter" type="text" /> <input id="gomb2" type="button" value="ok" />  
</form>
```

Most elindítva egy vízszintesen félbeosztott lapot láthatsz, aminek az alsó fele HTML, a felső része pedig Silverlight, amint azt a 9-23 ábra mutatja.



9-23 ábra: „Átlátszatlan” plug-in

Alkalmazásodnak futtatáskor paramétereket is átadhatsz. Erre való az `initParams` nevű paraméter, amint az alábbi példa is mutatja:

```
<param name="initParams" value="gyumolcs=alma,szin=piros" />
```

Az így megadott paramétereket a Silverlight alkalmazás a **Startup** eseményében kapja meg. Nyisd meg az **App.xaml.cs** fájlt, és keresd meg benne az **App_Startup** nevű metódust! Definiáld ezt az alábbi módon:

```
public string Gyumolcs;
public string Szin;

private void Application_Startup(object sender, StartupEventArgs e)
{
    if (e.InitParams.ContainsKey("gyumolcs"))
        this.Gyumolcs = e.InitParams["gyumolcs"];

    if (e.InitParams.ContainsKey("szin"))
        this.Szin = e.InitParams["szin"];

    this.RootVisual = new MainPage();
}
```

Nyisd meg a **MainPage.xaml.cs** fájlt, és módosítsd a konstruktorát:

```
public MainPage()
{
    InitializeComponent();
    textBlock1.Text = ((App)Application.Current).Gyumolcs;
    textBlock2.Text = ((App)Application.Current).Szin;
}
```

Elindítva láthatod, hogy a Silverlight felhasználja a HTML-be beírt adatokat.

Az itt tárgyalt paramétereken kívül még számos más paramétert is meg lehet adni a plug-innek, mint például a lokalizálásnál használt **uiculture** nevűt. Ezek összefoglaló listáját megtalálod az MSDN library-ban, vagy ha egyszerűen rákeresel az Interneten a „Silverlight Plugin Object Reference” kifejezésre.

JavaScript hívása Silverlightből

Silverlight kódból a HTML oldal elérése a **HtmlPage** osztály (**System.Windows.Browser** névtér) segítségével történik. Ennek **Window** tulajdonsága tartalmaz egy **Invoke** metódust, és ezzel a C# kódból közvetlen tudsz JavaScript funkciókat meghívni.

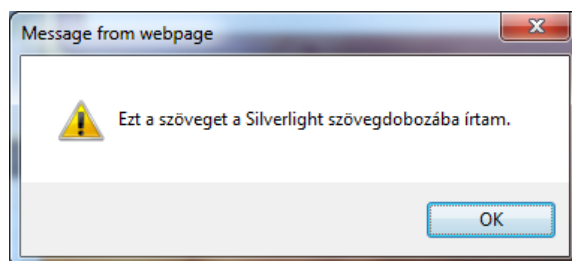
Írd be a HTML lapba a következő script blokkot közvetlenül a **</head>** elem elé:

```
...  
<script type="text/javascript">  
    function Figyelmeztetes(szoveg) {  
        alert(szoveg);  
    }  
</script>  
</head>
```

Ezt a funkciót a **MainPage** kódjából a MainPage ablakon lévő első gomb **Click** eseményében az alábbi módon hívhatod:

```
private void button1_Click(object sender, RoutedEventArgs e)  
{  
    HtmlPage.Window.Invoke("Figyelmeztetes", textBox1.Text);  
}
```

Az **Invoke** metódus első paramétere a JavaScript metódus neve, az utána felsorolt paramétereket pedig átadja sorrendhelyesen a hívott funkciónak. Indítás után az első Silverlight szövegdobozba írd valamit, és nyomd meg a mellette található gombot. Az eredmény egy böngésző dialógus lesz, amit a JavaScript metódus dobott, amint azt a 9-24 ábra mutatja.



9-24 ábra: A böngésző dialógusablaka

HTML manipulálása Silverlightből

A **HtmlPage** metódusait, tulajdonságait végignézve látható, hogy a böngésző és a HTML lap szinte minden porcikájához hozzá lehet férni a C# kódból, amint azt a következő példa is mutatja:

```
private void button2_Click(object sender, RoutedEventArgs e)  
{  
    HtmlElement gomb = HtmlPage.Document.GetElementById("gomb1");  
    if (gomb != null)  
        gomb.SetAttribute("value", textBox2.Text);  
  
    HtmlElement ujGomb = HtmlPage.Document.CreateElement("input");  
    ujGomb.SetAttribute("type", "button");  
    ujGomb.SetAttribute("value", "Dinamikus Gomb " + textBox2.Text);  
  
    HtmlPage.Document.Body.AppendChild(ujGomb);  
}
```

A **gomb** nevű változóban megkapod a **gomb1** HTML elemet a lapról. Ennek **value** attribútumát a szövegdozoz tartalmára állítod. A metódus végén egy új HTML gombot hozol létre dinamikusan, amit az utolsó sor a **body** végére illeszt.

XAML manipulálása JavaScriptből

Ez fordítva is működik: a Silverlight objektummodellje is elérhető a JavaScriptből. A beillesztett plug-in példánynak van egy **content** nevű tulajdonsága, amin keresztül elérhető a megjelenített tartalom. A Silverlight névvel ellátott objektumait el lehet érni a **content** tulajdonság **findName** metódusával azok neve alapján. Hogy ez kivitelezhető legyen, először a plug-in objektumra lesz szükséged, adj neki azonosítót:

```
<object id="silverlight" data="data:applic...
```

Írj egy új JavaScript funkciót a Figyelmeztetes funkció alá **XamlModositas** néven:

```
function XamlModositas() {
    var slh = document.getElementById("silverlight");
    var txt = document.getElementById("duma");
}
```

Most elkérheted a Silverlighttől a harmadik **TextBlock**-ot, a **findName** metódus segítségével, majd annak **Text** tulajdonságát be is állítod:

```
var textBlock = slh.content.findName('textBlock3');
textBlock.Text = txt.value;
```

Akár a XAML alapján teljesen új elemeket is beszúrhatsz a megjelenített Silverlight felületbe. Ehhez a **createFromXaml** metódus nyújt segítséget:

```
var stackPanel = slh.content.findName('stackPanel1');
var ellipse = slh.content.createFromXaml('<Ellipse Fill="#AAFF0000" Width="150"
    Height="20" />', false);
stackPanel.children.add(ellipse);
```

Ezt a metódust az első gombra kattintással kell meghívni:

```
<input type="button" id="gomb1" value="ok" onclick="XamlModositas();" />
```

Próbáld ki! Az első szövegdozozba írt szöveg a mellette található gomb megnyomásakor megjelenik a Silverlight harmadik szövegblokkjában, és egy kissé átlátszó ellipszis minden gombnyomás után hozzá adódik a **StackPanel**-hez.

Összefoglalás

Ebben a fejezetben megismerkedhettünk a Silverlight — talán aprónak tűnő, de mégis fontos — képességeivel, amelyek a böngészőben vagy azon kívül (OOB, out-of-browser) futó alkalmazások lehetőségeit gazdagítják. Mindezek segítségével a Silverlight olyan lehetőségekkel gazdagodik, amelyek a desktop alkalmazásoknál megszokott funkcionalitás elkészítésében segítik a fejlesztőket.

10. Üzleti alkalmazások fejlesztése Silverlight-ban

Az előző fejezetek mindegyike a Silverlight 4.0 alapú alkalmazás-fejlesztéssel foglalkozott, a fókusz pedig magára a platformra és a technológiai kérdésekre helyezte. Ez a tudás elengedhetetlen a most következő fejezetek megértéséhez, és az olvasottak elsajátításához. Ebben a fejezetben a prezentációs rétegben leggyakrabban használt tervezési minták alkalmazásáról lesz szó.

Üzleti alkalmazások fejlesztésének követelményei

Tesztelés és a szoftver minőség

Üzleti alkalmazások esetén a szoftver minősége kulcsfontosságú tényező. Az esetek nagy részében üzleti alkalmazások fejlesztése esetén a szoftverhez kapcsolódó igények már megfogalmazásra kerültek, így nem az értékesítés, az eladhatóság miatt kell aggódnunk. A felhasználó az alkalmazásunkat elsősorban a minősége alapján fogja megítélni. Ez persze így rendkívül tág és absztrakt fogalom, nem véletlenül számszerűsítik és konkretizálják ezeket a minőségi követelményeket. Ilyen minőségi követelmény lehet egy adott művelet meghatározott időn belüli végrehajtása. Szintén fontos minőségi követelmény, hogy a szoftver a lehető legkevesebb hibával rendelkezzen, és ha egy hiba mégis jelentkezik, akkor azt megfelelően kezelje.

A szoftver megfelelő minőségének biztosításához az egyik elengedhetetlen hozzávalója a tesztelés. Akármilyen alkalmazást is írunk, függetlenül a platformtól, a terméket tesztelni kell. A teszteket többnyire tesztmérnökök tervezik, és hajtják végre, de a fejlesztőkre is komoly munka hárul. Az általuk megírt komponenseket, kódokat áthatóan tesztelniük kell hagyományos működés és különböző marginális bemenetek esetén is. Ennek szellemében az ún. unit tesztek írása napi rutin kell, hogy legyen egy fejlesztő számára. Jelentős mennyiségű funkcionalitás és teszt birtokában, a tesztesetek állományának megléte azzal a jótékony mellékhatással jár, hogy az újabb vagy módosított, kiegészített funkciók működésének vizsgálata könnyedén elvégezhető a korábban megírt tesztek segítségével.

Annak ellenére, hogy a Silverlight kliensünk csupán prezentációs réteget képviseli, helyes működését tesztelnünk kell. Így **az első számú követelményünk a prezentációs réteg tesztelhetősége.**

Karbantarthatósági szempontok

Az idő előrehaladtával, a szoftver egyre nagyobb méreteket ölt. Ezzel jó esetben egyenesen arányosan, kevésbé jó esetben exponenciálisan nő a komplexitása is. Az alkalmazás egyre összetettebb, egyre nehezebben tekinthető át. Az alkalmazáson nem csupán egyetlen ember dolgozik, hanem egy egész csapat. A csapat tagjai időnként cserélődhetnek, érkezhetnek új emberek. Életbevágó szempont, hogy a szoftver architektúrája, de akár az implementációs részletei is könnyen áttekinthetők és érthetők legyenek. Nem szeretnénk, ha új emberünk napokat töltene azzal, hogy felderítse hogyan is működik alkalmazásunk. Nélkülözhetetlen embert pedig pláne nem szeretnénk csapatunkban tudni, nem számítani milyen zseniális Silverlight fejlesztő valaki, ha egyszer olyan kódot ír, amit más képtelen menedzselni, karbantartani. Szintén fontos szempont, hogy a módosításokat a lehető legkönnyebben legyünk képesek elvégezni. A kusza függőségek megléte a karbantarthatatlanság démonával fenyegethet. Ismerős az a kifejezés, hogy spagetti kód? A spagetti kód minden fejlesztő rémálma, amikor nem tudunk a tányérból egyetlen szál spagettit kihúzni anélkül, hogy a tányér tartalmának fele a fejünkön landolna. Így **a második számú követelményünk a karbantartható kód írása.**

A fejlesztő és a dizájnér közötti együttműködés

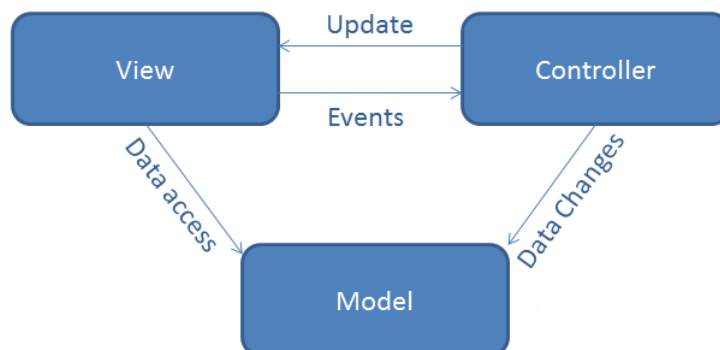
A Silverlight segítségével kétség kívül lenyűgöző felhasználói felületeket alakíthatunk ki. Ha nem tévedek, sokan egyetértenének velem abban a kijelentésben, hogy ezeket a lenyűgöző felhasználói felületeket a lehető legritkábban készítik fejlesztők. A Silverlight-os UI nem egy mozdulatlan, merev felület, egy igazi Silverlight-os UI „él”! A megfelelő kompozíciók pedig a dizájnerek agyában születnek meg. A manapság az üzleti alkalmazásokat is fel kell készíteni a mai világ igényeire, izléses, letisztult, intuitív felhasználói felületek kialakításával. A tökéletes összhanghoz azonban a dizájnér kezét nem szabad megkötnünk. A lehető legnagyobb szabadságot kell biztosítani számára. Ennek megfelelően a fejlesztő és a dizájnér közötti együttműködés a termék végkimenetele szempontjából kulcskérdés. A két szereplőnek egymástól függetlenül, egymással párhuzamosan kell tudnia dolgozni. A jól kialakított munkamenet és architektúra eredményeképp a végső felhasználói felület és hozzá kapcsolódó funkcionalitás illesztése pusztán az utolsó lépés kell legyen. Így a **harmadik követelményünk a fejlesztő és a dizájnér közötti hatékony, egészséges együttműködés kialakítása.**

Architekturális minták a prezentációs rétegben

A fenti követelmények kielégítésére megállapíthatjuk, hogy a felhasználói felületet le kell választani teljes mértékben a kapcsolódó logikáról és funkcionalitásról. Ez az igény nem új keletű és nem is a Silverlight / WPF megjelenésével merült fel. Ez az elvet hívják *separated presentation*-nek.

A Model-View-Controller architektúrális minta

A separated presentation elvnek számos konkrét megvalósítása létezik, ezek közül talán a legismertebb a Model-View-Controller (MVC) minta. Számos technológiában alkalmazzák előszeretettel, ezt a mintát. (pl. Java, WinForms, ASP.NET MVC, Ruby On Rails) Az MVC elsődleges feladata, hogy leválassza a felhasználói felületet (View), a kapcsolódó üzleti logikáról, adatforrásokról (Model) és a kettő közötti kapcsolatot és szinkronizációt a Controller segítségével oldja meg. A működés egyszerű. Ha a view-n bekövetkezik egy esemény, akkor jelezzük azt a controller felé, aki a Model-ben elvégzi a szükséges változtatásokat. Ha a Model-ben történik valami változás, akkor a Controller jelez a View-nak, hogy frissítse be magát. Az 10-1 ábra a három komponens közötti kapcsolatot tökéletesen ábrázolja. A Controller függ a Model-től és a View-től is, hiszen mindkettővel kommunikál, a View pedig függ a Controller-től és a Model-től is, hiszen a Controller-rel kommunikál, de az információ a Model-ből jön. A Model azonban nem kell hivatkoznia egyik komponensre sem, így az semmilyen módon sem kell, hogy függjön a felhasználói felülettől. A Controller és a modell is pompásan tesztelhető önállóan, és a View-Controller párosnak köszönhetően a karbantarthatóság terén is komoly lépést tettünk előre.



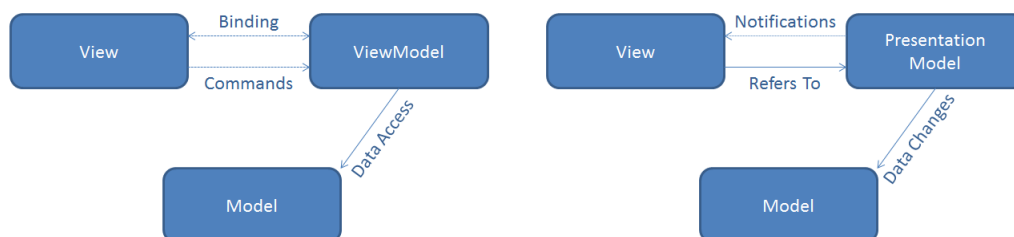
10-3 ábra – Az MVC architektúra

Az MVC minta azonban továbbra is elég magasan tartja a függőségek számát, ugyanakkor tény, hogy mindezt jóval kezelhetőbb formában teszi. A Silverlight adatkötési modelljének köszönhetően azonban a lazán csatolás lehetősége egy új, az MVC gondolatvilágára épülő, némileg modernebb minta kialakulását tette lehetővé.

A Model-View-ViewModel architektúrális minta

A Model-View-ViewModel (MVVM) architektúrális mintát a Microsoft vezette be kimondottan a WPF és a Silverlight számára. Ez a minta is természetesen az MVC elveit követi. Az MVVM-et John Gossman Silverlight és WPF architekt jeleltette be 2005-ben, és úgy nevezte meg a technológiát, mint Martin Fowler Presentation Model-jének a WPF újszerű képességeihez igazított specializálását.

A 10-2 ábrán látható az MVVM és a Presentation Model elvek összehasonlítása.



10-4. ábra – Az MVVM és a Presentation Model architektúrális minták összehasonlítása

A **Model** alatt, akár csak MVC-ben, a különböző entitás osztályok, adatforrások, DTO-k, proxy kliensek, és egyéb hasonló, a prezentációtól független komponensek együttesét értjük. Akár adatok mentésére, akár betöltésére van szükség, ezt a Modellen keresztül lehet végrehajtani.

A **View** alatt a felhasználói felület leírását értjük. Sokan úgy interpretálják az MVVM-et, hogy ebben a tervezési mintában nem lehet mögöttes kódot írni. Azonban ez nem igaz, az MVVM egy architektúrális minta, és semmilyen ilyen jellegű megkötést nem tartalmaz. Akár WPF akár Silverlight esetén előfordul, hogy kénytelenek vagyunk a mögöttes állományban kódot elhelyezni, azonban ez kizárólag a megjelenítéssel és a felhasználói felülettel kapcsolatos, ún. UI specifikus kód lehet.

A **ViewModel** alatt a View aktuális állapotának leírását értjük. A ViewModel-nek összetett feladata van. Egyrészt a Model-ből elő kell állítani a szükséges adatokat, illetve a View-n keletkezett eseményekre, melyek a ViewModel oldalán funkcióhívásként jelentkeznek, megfelelően kell tudni reagálni, a View-n keletkezett inputokat kezelnie kell. Másrészt a ViewModel egyben a felhasználói felülethez kapcsolódó állapotokat is tartalmazza. Így jelezhet a View-nak olyan egyéb információkat, minthogy a háttérben éppen adatok betöltése zajlik, egy panel logikai szempontból látható-e vagy sem, a View-n megjelenített adat érvényes-e vagy sem és így tovább.

Az MVVM tervezési mintában a nagyszerű újdonság a binding engine-nek köszönhető. A View és a ViewModel illesztése történhet lazán csatoltan, anélkül, hogy a View ismerné a ViewModelt vagy a ViewModel ismerné a View-t. Vagyis a két komponens egymástól függetlenül fejlődhet. Az MVVM architektúrális minta alkalmazásának köszönhetően egy tesztelhető, jól karbantartható alkalmazást kapunk, ahol a View és a ViewModel fejlesztése párhuzamosan történhet a fejlesztő és a dizájnner munkájának eredményeképp.

Sztori: A Visual Studio 2010 Magyarországi bejelentéséhez kapcsolódó konferencián az egyik előadásom a felhasználói felületek teszteléséről szólt. Az előadást követően két fiatal fejlesztő megkeresett, tanácsot kértek. Windows Forms-os projekten dolgoztak és a következő problémával találkoztak szembe:

Ahogy nőtt az alkalmazásuk, egyre nehezebben tudták karbantartani. Ha a megrendelő részéről jött egy új igény, vagy egy hibát kellett javítani, a módosítások időnként dominószerűen sok mindent felborítottak az alkalmazás működésében. Így elérkezettnek látták az időt, hogy elkezdjenek teszteket írni. Jelen esetben számos unit teszt írására lett volna szükség. Az igazi nagy probléma akkor jött elő, mikor kiderült, hogy a felhasználói felülettel a logika nagyon szorosan összenőtt. Az eseménykezelők kódjai hosszúak és komplexek voltak, a felhasználói felülettel pedig számos komponens interakcióba lépett. Leválasztani és függetleníteni a felhasználói felület a kódtól óriási munkát és újra írást jelentett volna.

Ismerősen hangzik? A rendkívül magas számú függőségek miatt utólagosan sem a módosítást, sem a unittesztelést már nem tudták bevezetni, így próbálták a károkat és a problémákat enyhíteni UI tesztelés bevezetésével. Mindez elkerülhető lett volna, hogyha már kezdettől fogva alkalmazták a Separated Presentation elv valamilyen implementációját.

A MVVM architektúrális minta gyakorlati alkalmazása

MVVM keretrendszerek használata

Az MVVM architektúrális minta nincs kőbe vésve. Nincs egy Microsoft által kibocsátott ajánlás csomag, és kódhalmaz, ami meghatározná és keretek közé zárná, hogy pontosan hogyan is kell az MVVM-et implementálni. Így nyugodtan építhetünk saját megoldásokat. Azonban hamar rájövünk, hogy sokszor futunk ugyanabba a problémába a mintával kapcsolatban. Ezekre a problémákra nem kell saját megoldásokat kitalálnunk, nyugodtan használhatunk, mások által kifejlesztett MVVM keretrendszereket. Jelen pillanatban a két leghíresebb keretrendszer az MVVM Light Toolkit és a Prism.

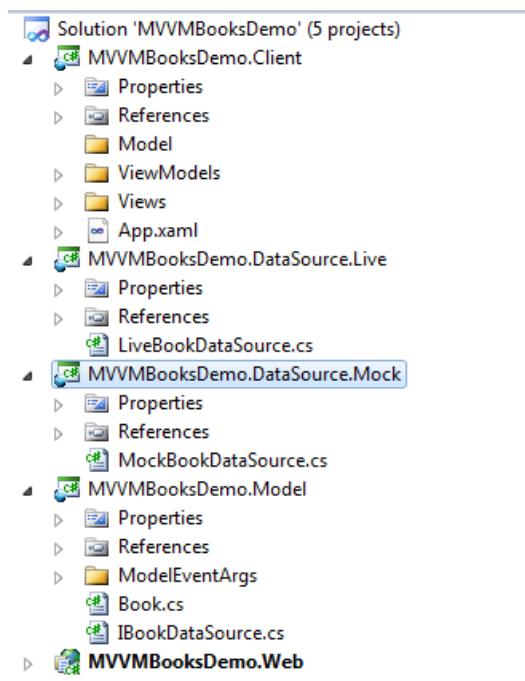
- **Az MVVM Light Toolkit** egy nyílt forráskódú MVVM keretrendszer, melyet Laurent Bugnion Silverlight MVP fejleszt folyamatosan. A toolkitnek van WPF 3.5, 4.0, Silverlight 3.0 és 4.0, illetve Windows Phone 7 változata is. A legalapvetőbb problémákra próbál megoldást kínálni, mint az ún. *commanding* támogatása, design time támogatás, vagy a View-k közötti lazán csatolt kommunikáció. Ezen kívül egyéb hasznos, MVVM-ben jól használható behavior-öket is tartalmaz.
- A **Prism**, vagy teljes nevén Composite Application Guidance egy a Microsoft Patterns and Practices csapata által fejlesztett nyílt forráskódú osztálykönyvtár. A Prism akárcsak az MVVM Light Toolkit szintén az MVVM tervezési mintát támogatja, de a fő képessége a moduláris alkalmazások fejlesztésének támogatása. Az osztálykönyvtárat folyamatosan fejlesztik, a könyv írásának pillanatában a Prism 4.0-ás változata elérhető. Erről bővebben a 12-ik fejezetben olvashatunk.

Az alkalmazás struktúrájának kialakítása

Az MVVM-mel történő ismerkedés során egy példa alkalmazást fogunk folyamatosan fejleszteni és új képességekkel felruházni. A fogorvosi lovunk egyszerű, könyveket karbantartó alkalmazás lesz. Az alkalmazás az adatokat a fejlesztési ideje alatt szimulált adatforrásból fogja betölteni.

A projekt struktúrája a 10-3-as ábrán látható. A hagyományoknak megfelelően alakítottuk ki a mappa struktúrát az MVVM számára. A nézetek külön Views mappát kaptak. Ebben a mappába két UserControl került felvételre. Bevezető példaként Master-Details szituációt fogunk kialakítani. A Master szerepét **BookView** UserControl tölti majd be, míg az aktuálisan kiválasztott könyv részleteit **BookDetailsView** UserControl jeleníti meg.

Bár hagyományosan a **Model** mappát is elkészítettük, az entitás osztályokat és az adatforrásokat ezúttal egy külön projekt-be, az **MVVMProductsDemo.Model**-be szerveztük.



10-3 ábra: A projekt struktúrája

A Model projekt

A **Model** projektben szerepel a **Book.cs** fájl, ami a **Book** osztályt reprezentálja. Ezt az osztályt fogjuk majd használni a könyvek leírására a kliens és a szerver oldalon is. Minden entitást és adatforrás absztrakciót, ebben a projektbe helyezünk el.

A **Book** osztály felépítése az alábbi:

```
public class Book
{
    public int BookID { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public decimal Price { get; set; }
}
```

Az **IBookDataSource** interfészen keresztül léphet majd a ViewModel interakcióba az adatforrással, akár szimulált, akár éles adatforrásról beszélünk. A kommunikáció jellege teljes mértékben aszinkron, így ennek megfelelően a könyvek listáját betöltő **LoadBooks()** metódushoz kapcsolódik egy **LoadBooksCompleted** esemény, a kiszemelt könyv törléséért felelős **RemoveBook(int bookID)**, metódushoz pedig egy **RemoveBookCompleted** esemény.

Az adatforrásokat reprezentáló **IBookDataSource** interfész kódja az alábbi:

```
public interface IBookDataSource
{
    void LoadBooks();
    void RemoveBook(int bookID);

    event EventHandler<LoadBooksCompletedEventArgs> LoadBooksCompleted;
    event EventHandler<OperationCompletedEventArgs> RemoveBookCompleted;
}
```

Az eredmény nélküli aszinkron hívások **EventArgs** osztálya:

```
public class OperationCompletedEventArgs : EventArgs
{
    public Exception Error { get; set; }
}
```

A könyvek betöltéséhez kapcsolódó **EventArgs** osztály:

```
public class LoadBooksCompletedEventArgs : OperationCompletedEventArgs
{
    public IEnumerable<Book> Books { get; set; }
}
```

A DataSources.Mock projekt

Ahhoz, hogy legyen egy működő adatforrásunk, az **IBookDataSource** interfészt kell implementálni egy osztályban. Ez meg is teszi a **MockBookDataSource** osztály, amely egy könyv adatforrást szimulál. Minden Mock jellegű adatforrást, a **DataSources.Mock** projektben helyezünk el.

A **MockBookDataSource** osztály egy lehetséges implementációját mutatja be az alábbi kód:

```
public class MockBookDataSource : IBookDataSource
{
    //Mock könyv adatforrás
    List<Book> mockBooks = new List<Book>();

    public MockBookDataSource()
    {
        //adatforrás feltöltése példaadatokkal
        mockBooks.Add(
            new Book
            {
                BookID = 1,
                Title = "Executive Orders",
                Author = "Tom Clancy",
                Price = 14.99m
            });

        mockBooks.Add(
            new Book
            {
                BookID = 2,
                Title = "The DaVinci Code",
                Author = "Dan Brown",
                Price = 19.99m
            });

        mockBooks.Add(
            new Book
            {
                BookID = 3,
                Title = "The Bourne Ultimatum",
                Author = "Robert Ludlum",
                Price = 9.99m
            });
    }
}
```

```

public void LoadBooks()
{
    // Jelezzük, hogy a könyvek betöltése sikeres, majd küldjük el az
    // EventArgs-ban könyveket
    Thread.Sleep(2000); // szimuláljuk az aszinkron késleltetést
    if (LoadBooksCompleted != null)
        LoadBooksCompleted(this,
            new LoadBooksCompletedEventArgs { Books = mockBooks });
}

public void RemoveBook(int bookID)
{
    Thread.Sleep(2000); //szimuláljuk az aszinkron késleltetést
    Exception error = null;

    // Létezik ilyen könyv? Ha nem, akkor error példány beállít
    if (mockBooks.Where(p => p.BookID == bookID).Count() == 0)
    {
        error = new ArgumentException("There is no book with this ID");
    }

    // A RemoveBook művelet befejeződött, ha volt hiba, az Error tulajdonság
    // nem null értékű
    if (RemoveBookCompleted != null)
        RemoveBookCompleted(this, new OperationCompletedEventArgs { Error = error });
}

public event EventHandler<Model.ModelEventArgs.LoadBooksCompletedEventArgs>
    LoadBooksCompleted;

public event EventHandler<Model.ModelEventArgs.OperationCompletedEventArgs>
    RemoveBookCompleted;
}

```

A DataSources.Live projekt

Minden olyan implementációt, amely éles adatforrásból tölti majd az adatokat, a **DataSources.Live** projektben helyezzük el.

A ViewModel-ek alapfunktionalitásának implementálása

Az MVVM-mel történő munkához szükség lesz egy segédkönyvtárra, amely segít nekünk a minta implementálásában. Ehhez az *MVVM Light Toolkit*-et választjuk. Az MVVM Light Toolkit telepítéséről részletes információkat a toolkit honlapján találunk, ugyanakkor jelen példában az is bőven elegendő, hogy ha a toolkit-ben található **Galasoft.MVVMLightToolkit.dll**-re felveszünk egy referenciát.

A ViewModelek feladata tehát, hogy a View számára biztosítsák azon információkat, amelyek a megjelenéssel kapcsolatosak. Minden ilyen tulajdonság, funkció a **BooksViewModel** osztályba fog kerülni.

Az MVVM Light Toolkit használata esetén a ViewModel-jeink a **ViewModelBase**-ből származnak. Ez az őszintáz számos kényelmi funkciót biztosít számunkra. Így például segítségével eldönthetjük, hogy tervezési időben dolgozunk, vagy ténylegesen fut az alkalmazásunk, illetve az őszintáz már implementálja az **INotifyPropertyChanged** interfészt is. Az esemény kiváltása pedig a **RaisePropertyChanged(string propertyName)** metódus segítségével történhet.

A fentieknek megfelelően, ha a könyvek listáját szeretnénk megjeleníteni, szükség lesz egy gyűjteményre, amely a betöltött könyveket fogja reprezentálni. Ez lesz a **Books** gyűjtemény, ami történetesen egy **ObservableCollection<BookDetailsViewModel>** típusú objektum.

Jogosan merül fel a kérdés, hogy ezek szerint minden entitás osztály felé szükséges egy ViewModel-t készíteni? Miért nem elég az **ObservableCollection<Book>**? Erre a kérdésre a választ az alábbiakban adhatjuk meg:

1. A részletező nézetet a listában kiválasztott elemhez kell kötnöm. Így mindig az aktuálisan kiválasztott elem részleteit jeleníthetjük meg a részletező nézetben. A részletező nézetnek nem megfelelő, ha a saját **DataContext**-je **Book**. A **Book** osztály önmagában csak adatot hordoz. A részletező nézetnek azonban funkcionalításra is szüksége van. A **Book** osztály ilyen jellegű kiterjesztése tervezési szempontból nem helyes. Ezért egy **BookDetailsViewModel** osztályba csomagolom a **Book** példányt. Ebben az esetben a részletező nézetet nem lehet egy **ObservableCollection<Book>** kiválasztott eleméhez kötni, muszáj, hogy a listában csupa **BookDetailsViewModel** legyen.
2. Ha a részletező nézetet nem is vesszük figyelembe, képzeljük el a következő szituációt. Egy **DataGrid** vezérlőben jelenítjük meg a könyvek egy listáját. Minden egyes sor előtt elhelyezünk egy Remove gombot. Ha erre a gombra ráklikkelünk, a kapcsolódó elemet töröljük a listából. Ebben az esetben is hasonló problémába ütközünk, mint az előző pontban. A **BookDetailsViewModel** a megfelelő hely extra funkciók felvételére, így a törlésre is.
3. Szeretnénk a függőséget a Model és a nézet között csökkenteni.

Az MVVM minta ezt a kérdést sem tisztázza. A fejlesztőktől függ, hogy miképp döntenek, hogyan implementálják a mintát. Azt azonban látni kell, hogy minden egyes entitás fölé egy ViewModel-t építeni fáradságos dolog, sok extra munkát igényel. Ökölszabályként követhetjük a következő elvet: akkor készítsünk ViewModel objektumot az entitás osztály fölé, ha szükségünk van extra, az eredeti entitás osztályhoz nem tartozó funkciókra és tulajdonságokra.

Fontos: (Ez a szabály a szerző elképzelése az MVVM hatékony és célszerű implementálásáról, számos fejlesztő hisz a „tisztá” MVVM-ben és definiál minden entitás osztály fölé megfelelő ViewModel osztályt)

A **BookDetailsViewModel** osztály lehetséges implementációja:

```
public class BookDetailsViewModel : ViewModelBase
{
    IBookDataSource dataSource;
    Book book;

    public BookDetailsViewModel(Book book, IBookDataSource dataSource)
    {
        this.book = book;
        this.dataSource = dataSource;
    }

    public int BookID
    {
        get { return book.BookID; }
    }

    public string Title
    {
        get { return book.Title; }
        set
        {
            if (book.Title == value) return;
            book.Title = value;
            RaisePropertyChanged("Title");
        }
    }
}
```

```

public string Author
{
    get { return book.Author; }
    set
    {
        if (book.Author == value) return;
        book.Author = value;
        RaisePropertyChanged("Author");
    }
}

public decimal Price
{
    get { return book.Price; }
    set
    {
        if (book.Price == value) return;
        book.Price = value;
        RaisePropertyChanged("Price");
    }
}

public void RemoveBook()
{
    //Könyv törlése az adatforrásból
    dataSource.RemoveBook(BookID);
}
}

```

A **BooksViewModel** osztály első változata:

```

public class BooksViewModel : ViewModelBase
{
    // Adatforrás
    IBookDataSource dataSource = null;

    private ObservableCollection<BookDetailsViewModel> books = new
    ObservableCollection<BookDetailsViewModel>();

    public ObservableCollection<BookDetailsViewModel> Books
    {
        get { return books; }
        set { books = value; }
    }

    public BooksViewModel()
    {
        if (IsInDesignMode)
        {
            //Design-time-ban design adatokkal dolgozunk
            LoadDesignData();
        }
        else //Egyébként éles adatokkal dolgozunk
        {
            dataSource = new MockBookDataSource();
            dataSource.LoadBooksCompleted +=
                new EventHandler<Model.ModelEventArgs.LoadBooksCompletedEventArgs>(
                    dataSource_LoadBooksCompleted);
            dataSource.RemoveBookCompleted +=
                new EventHandler<Model.ModelEventArgs.OperationCompletedEventArgs>(
                    dataSource_RemoveBookCompleted);
            // Könyvek automatikus betöltése
            LoadBooks();
        }
    }
}

```

```
}

void dataSource_RemoveBookCompleted(object sender,
Model.ModelEventArgs.OperationCompletedEventArgs e)
{
    if (e.Error != null)
    {
        //Logoljuk és jelenítsük meg a hibát
    }
}

void dataSource_LoadBooksCompleted(object sender,
Model.ModelEventArgs.LoadBooksCompletedEventArgs e)
{
    if (e.Error != null)
    {
        //Logoljuk és jelenítsük meg a hibát
    }
    else
    {
        //Töröljük a könyvlistát a memóriából
        Books.Clear();

        //Újra feltöltjük a listát
        foreach (Book book in e.Books)
        {
            BookDetailsViewModel bookDetailsViewModel =
                new BookDetailsViewModel(book, dataSource);
            Books.Add(bookDetailsViewModel);
        }
    }
}

// Adatok betöltése az adatforrásból
public void LoadBooks()
{
    dataSource.LoadBooks();
}

// Design-time adatok betöltése
private void LoadDesignData()
{
    // Töltsünk design-time adatokat a Books gyűjteménybe...
    List<Book> designTimeBooks = new List<Book>();
    designTimeBooks.Add(
        new Book
        {
            BookID = 1,
            Title = "Executive Orders",
            Author = "Tom Clancy",
            Price = 14.99m
        });

    designTimeBooks.Add(
        new Book
        {
            BookID = 2,
            Title = "The DaVinci Code",
            Author = "Dan Brown",
            Price = 19.99m
        });
}
```

```

        designTimeBooks.Add(
            new Book
            {
                BookID = 3,
                Title = "The Bourne Ultimatum",
                Author = "Robert Ludlum",
                Price = 9.99m
            });

        foreach (Book book in designTimeBooks)
        {
            BookDetailsViewModel bookDetailsViewModel =
                new BookDetailsViewModel(book, dataSource);
            Books.Add(bookDetailsViewModel);
        }
    }
}

```

Természetesen, szeretnénk nyomon követni az aktuálisan kiválasztott elemet is, hiszen az ehhez kapcsolódó részleteket és funkcionalitást a részletező nézetben szeretnénk bekötni! Konkretizálva ez azt jelenti, hogy a **ListBox SelectedItem** tulajdonságát és a **BookDetailsView** vezérlő **DataContext** tulajdonságát is a **SelectedBook** tulajdonsághoz szeretnénk adatkötni. Miért van erre a köztes tulajdonságra szükség? Miért nem elegendő UI-to-UI binding-ot alkalmazni? Erre számos okunk lehet. Ezek közül néhány:

1. Az alkalmazás többi részét értesíteni szeretnénk arról, ha a kiválasztott könyv megváltozik. Ezt a változást — a vezérlők tulajdonságainak közvetlen összekötése esetén — nincs lehetőségünk jelezni. Azonban, ha az összekötés egy ViewModel-ben található köztes tulajdonságon át (**SelectedBook**) történik, akkor ennek a tulajdonságnak setter metódusában már bármit megtehetünk!
2. A kiválasztott könyvet szeretnénk kódból módosítani. Például, a betöltést követően az első könyv legyen a kiválasztott elem.
3. A dizájnernek nem kell tudnia, hogy a két vezérlőelem szinkronizálása fontos funkcionális elem.

A **BooksViewModel** osztályt az alábbi módon változtathatjuk meg, hogy az a kiválasztott könyvet is számontartsa:

```

//Új tulajdonság felvétele
private BookDetailsViewModel selectedBook;

public BookDetailsViewModel SelectedBook
{
    get { return selectedBook; }
    set
    {
        if(selectedBook == value) return;
        selectedBook = value;
        RaisePropertyChanged("SelectedBook");
    }
}

// LoadBooksCompleted eseménykezelő módosítása, betöltés után az első
// könyv a kiválasztott
void dataSource_LoadBooksCompleted(object sender,
    Model.ModelEventArgs.LoadBooksCompletedEventArgs e)
{
    ...
    ...
    ...
}

```

```
        foreach (Book book in e.Books)
        {
            BookDetailsViewModel bookDetailsViewModel =
                new BookDetailsViewModel(book, dataSource);
            Books.Add(bookDetailsViewModel);
        }
    }

    SelectedBook = Books.First();
}

// Design-time adatok módosítása, az első könyv a kiválasztott
private void LoadDesignData()
{
    ...
    ...
    ...
    foreach (Book book in designTimeBooks)
    {
        BookDetailsViewModel bookDetailsViewModel = new BookDetailsViewModel(book, dataSource);
        Books.Add(bookDetailsViewModel);
    }
    SelectedBook = Books.First();
}
```

A ViewModel-jeinknek köszönhetően, mind tervezési időben, mind pedig futásidőben készen állunk a könyvek betöltésére. Amint a **BooksViewModel** osztályból létrejön egy példány, az adatok betöltése máris megkezdődik. Ez annak köszönhető, hogy a **BooksViewModel** konstruktorában indítjuk el az adatok betöltését, itt hívjuk meg a **LoadBooks()** metódust. Nincs más hátra, mint implementálni a felhasználói felületet.

A nézetek (Views) implementálása

A nézetek egyszerű **UserControl**-ok lesznek, kialakításuk ennek megfelelően történik. A **BooksView** nézet egy **ListBox**-ot tartalmaz, amiben majd a könyvek listáját jelenítjük meg, illetve egy **BookDetailsView UserControl**-t, amiben mindig a kiválasztott könyv részletei jelennek meg:

```
<UserControl
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:MVVMBooksDemo.Client.Views"
    x:Class="MVVMBooksDemo.Client.Views.BooksView"
    mc:Ignorable="d"
    d:DesignHeight="480" d:DesignWidth="640">

    <Grid x:Name="LayoutRoot" Background="White">
        <ListBox Margin="11,41,0,0" HorizontalAlignment="Left" Width="160" Height="260"
            VerticalAlignment="Top"/>
        <local:BookDetailsView Margin="198,40,0,0" Height="260" Width="300"
            HorizontalAlignment="Left" VerticalAlignment="Top"/>
    </Grid>
</UserControl>
```

A **BookDetailsView** felépítése az alábbi:


```

<UserControl
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  x:Class="MVVMBooksDemo.Client.Views.BookDetailsView"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="400">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="0.1*" />
      <RowDefinition Height="0.1*" />
      <RowDefinition Height="0.1*" />
      <RowDefinition Height="0.1*" />
      <RowDefinition Height="0.25*" />
      <RowDefinition Height="0.1*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="0.363*" />
      <ColumnDefinition Width="0.637*" />
    </Grid.ColumnDefinitions>
    <TextBlock TextWrapping="Wrap" Text="Book ID" d:LayoutOverrides="Width, Height"
      HorizontalAlignment="Left" VerticalAlignment="Center" />
    <TextBlock TextWrapping="Wrap" Text="Title" Grid.Row="1"
      d:LayoutOverrides="Width, Height" HorizontalAlignment="Left"
      VerticalAlignment="Center" />
    <TextBlock TextWrapping="Wrap" Text="Author" Grid.Row="2"
      d:LayoutOverrides="Width, Height" HorizontalAlignment="Left"
      VerticalAlignment="Center" />
    <TextBlock TextWrapping="Wrap" Text="Price" Grid.Row="3"
      d:LayoutOverrides="Width, Height" HorizontalAlignment="Left"
      VerticalAlignment="Center" />
    <TextBlock TextWrapping="Wrap" Grid.Column="1" d:LayoutOverrides="Height"
      VerticalAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="1" TextWrapping="Wrap"
      VerticalAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="2" TextWrapping="Wrap"
      VerticalAlignment="Center" />
    <TextBox Grid.Column="1" Grid.Row="3" TextWrapping="Wrap"
      VerticalAlignment="Center" />
    <Button Content="Remove" Grid.Column="1" HorizontalAlignment="Right"
      Margin="0,-1,0,1" Grid.Row="5" />
  </Grid>
</UserControl>

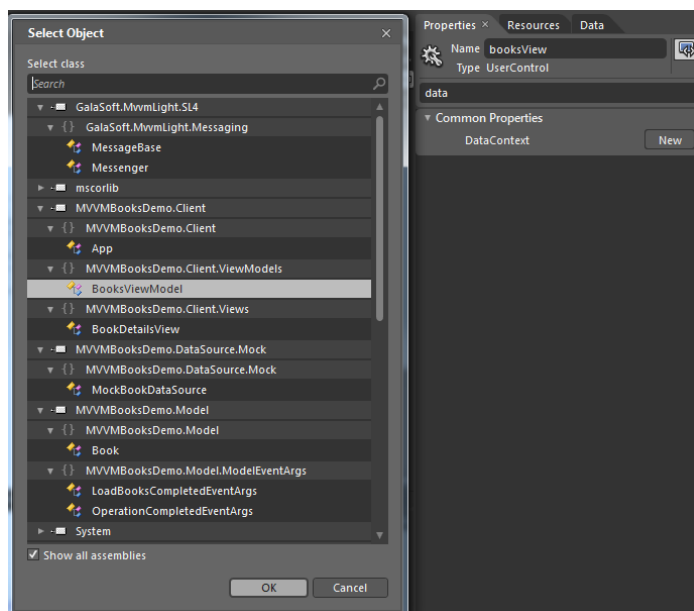
```

A nézetek és a ViewModel-ek összekötése

Korábbi munkánk gyümölcsét igazán most élvezhetjük! Jelen pillanatban még semmilyen adat sem került bekötésre. Azonban, az adatkötésben és az adat sablonok testre szabásában rendkívül sokat fog a Blend segíteni.

A Books gyűjtemény bekötése a ListBox-ba

A **BooksView** UserControl **DataContext**-jét kell a **BooksViewModel** objektumon beállítani. Ehhez az Expression Blend-ben az oldal **DataContext** tulajdonságánál kattintsunk a New gombra, majd a felugró panelben válasszuk ki a **BooksViewModel** objektumot. Kattintsunk az OK gombra (10-3 ábra).



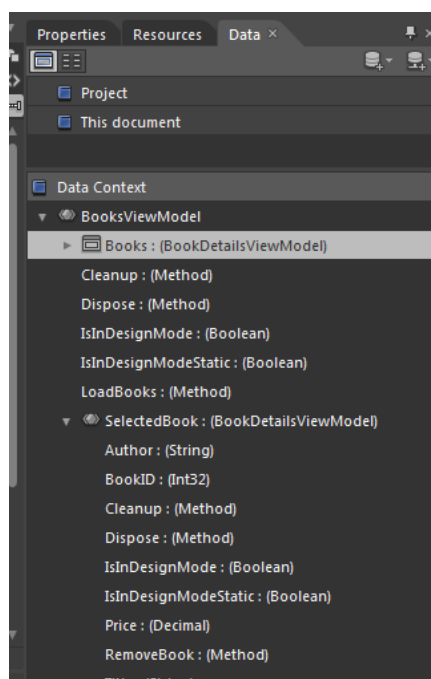
10-3 ábra: DataContext beállítása a ViewModel példányra

Ezt követően a **BooksView** vezérlő **DataContext**-jét a Blend beállítja egy új **BooksViewModel** példányra:

```
<UserControl.DataContext>
  <MVVMBooksDemo_Client_ViewModels:BooksViewModel/>
</UserControl.DataContext>
```

A következő lépésben a **ListBox ItemsSource** tulajdonságát kell a **Books** listához adatkötnünk. Ez történhet a már ismer módokon, Visual Studio 2010, illetve Blend, vagy akár a XAML editor segítségével is. Kétségen kívül, a **ViewModel** objektumok helyes bekötése a legpraktikusabban és legegyszerűbben a Blend **Data** paneljének segítségével oldható meg.

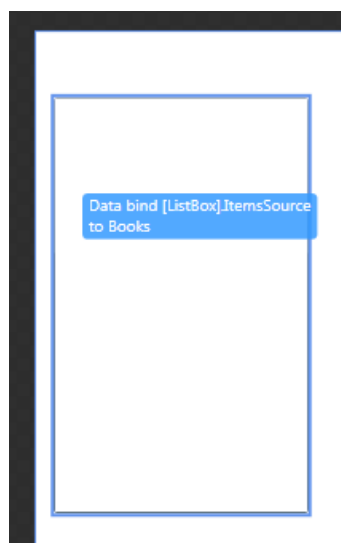
Az előző akció eredményeképp ugyanis ez a **Data** panel életre kelt a Blend-ben, ahogyan azt a 10-4 ábra is mutatja.



10-4 ábra: A Data panel adatkötés közben

A Data panel **DataContext** szekciója jól mutatja, hogy a bekötött ViewModel milyen tulajdonságokkal rendelkezik. Az egyes tulajdonságokat drag-and-drop megközelítéssel lehet az egyes vezérlőkhöz kötni. Így a **Books** gyűjteményt a **ListBox** fölé húzva a Blend figyelmeztet arra, hogy a **ListBox ItemsSource** tulajdonságát most a kérdéses **Books** listához fogja kötni (10-5 ábra).

Megjegyzés: A Blend ezzel egy időben a **ListBox ItemTemplate** tulajdonságát is beállítja az egyes elemek típusa alapján.



10-5 ábra: A Books lista hozzákötése a ListBox-hoz drag-and-drop közben

A Blend az alábbi kódot generálta ListBox-hoz:

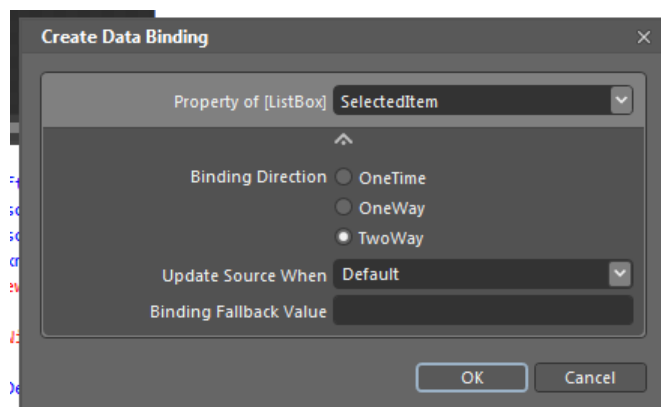
```
<ListBox Margin="11,41,0,0" HorizontalAlignment="Left" Width="160" Height="260"
    VerticalAlignment="Top"
    ItemTemplate="{StaticResource BookDetailsViewModelTemplate}"
    ItemsSource="{Binding Books}" SelectedItem="{Binding SelectedBook, Mode=TwoWay}"/>
```

A ListBox-hoz az alábbi **ItemTemplate**-et szintén a Blend állította elő:

```
<UserControl.Resources>
    <DataTemplate x:Key="BookDetailsViewModelTemplate">
        <StackPanel>
            <TextBlock Text="{Binding Author}"/>
            <TextBlock Text="{Binding BookID, Mode=OneWay}"/>
            <CheckBox IsChecked="{Binding IsInDesignMode, Mode=OneWay}"/>
            <CheckBox IsChecked="{Binding IsInDesignModeStatic, Mode=OneWay}"/>
            <TextBlock Text="{Binding Price}"/>
            <TextBlock Text="{Binding Title}"/>
        </StackPanel>
    </DataTemplate>
</UserControl.Resources>
```

A SelectedBook bekötése és szinkronizálása a két vezérlő között

Következő lépésben a **ListBox SelectedItem** tulajdonságát kell a ViewModel **SelectedBook** tulajdonságához kötnünk **TwoWay** módon. Ismét a Data panelről célszerű behúznunk a ListBox fölé a tulajdonságot. Ezúttal a Blend nem elég okos, és a ListBox DataContext-jére szeretni kötni a tulajdonságot. Azonban, ha a Shift gomb nyomvatartása mellett engedjük el az egér gombját a Blend egy kis dialógus ablakban (Create Data Binding ablak) engedi testreszabni az adatkötést, ahogyan azt a 10-6 ábra mutatja.



10-6 ábra: Shift lemonyása melletti drag-and-drop esetén manuálisan testreszabható az adatkötés

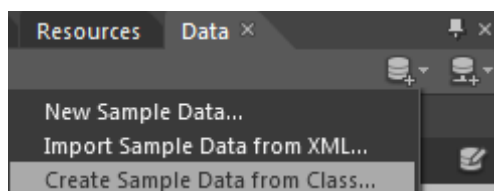
A **BookDetailsView** UserControl-ra drag-and-drop művelettel a **SelectedBook** tulajdonságot ráhúzva a Blend beköti a **DataContext** tulajdonságot:

```
<ListBox Margin="11,41,0,0" HorizontalAlignment="Left" Width="160" Height="260"
    VerticalAlignment="Top"
    ItemTemplate="{StaticResource BookDetailsViewModelTemplate}"
    ItemsSource="{Binding Books}" SelectedItem="{Binding SelectedBook, Mode=TwoWay}"/>
<local:BookDetailsView Margin="198,40,0,0" Height="260" Width="300"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    DataContext="{Binding SelectedBook}"/>
```

A BookDetailsViewModel tulajdonságainak bekötése a BookDetailsView UserControl-ban

A **BookDetailsView** belső részleteit szerkesztve sajnos a korábbi ViewModel-be épített tervezésidejű támogatásunk nem elérhető. Ha manuálisan bekötjük az egyes mezőket, majd visszalépünk a **BooksView** szintjére, a támogatást azonnal visszkapjuk. A **BookDetailsView** szerkesztése és egyes mezőinek bekötése közben ez nem nagy vigasz. Szerencsére az Expression Blend támogatja a ViewModel-ekből készített példaadatokat.

A **BookDetailsView** UserControl szerkesztése közben a Data panelen két kis „henger” ikont találunk. Az első ikon a SampleDataSource ikonja. Ezzel már korábbi fejezetekben találkozhattunk. Az ikonra kattintva a harmadik menüpont a „Create Sample Data from Class” nevet viseli (10-7 ábra). A segítségével a ViewModel objektumunk fölé készít egy tervezési időben használható adatforrást.



10-7 ábra: Példa adatforrás készítése ViewModel osztályból

A felugró ablakból a **BookDetailsViewModel1**-t kell választanunk. A Data panel-en akárcsak korábban megjelennek az adott ViewModel tulajdonságai, amelyek drag-and-drop módszerrel lehet bekötni. A példa adatforrás egyes mezőit igény szerint testreszabhatjuk.

A Blend az alábbi kódot generálja:

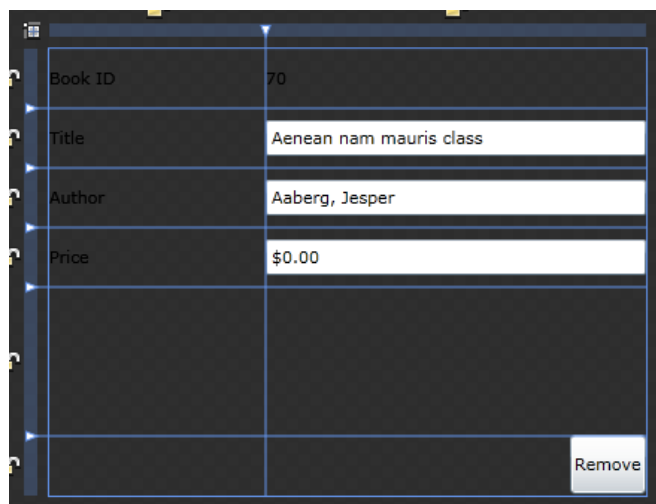
```

<UserControl x:Class="MVVMBooksDemo.Client.Views.BookDetailsView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">
    <Grid d:DataContext="{d:DesignData /SampleData/BookDetailsViewModelSampleData.xaml}">
        <Grid.RowDefinitions>
            <RowDefinition Height="0.1*" />
            <RowDefinition Height="0.1*" />
            <RowDefinition Height="0.1*" />
            <RowDefinition Height="0.1*" />
            <RowDefinition Height="0.25*" />
            <RowDefinition Height="0.1*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.363*" />
            <ColumnDefinition Width="0.637*" />
        </Grid.ColumnDefinitions>
        <TextBlock TextWrapping="Wrap" Text="Book ID" d:LayoutOverrides="Width, Height"
            HorizontalAlignment="Left" VerticalAlignment="Center" />
        <TextBlock TextWrapping="Wrap" Text="Title" Grid.Row="1"
            d:LayoutOverrides="Width, Height" HorizontalAlignment="Left"
            VerticalAlignment="Center" />
        <TextBlock TextWrapping="Wrap" Text="Author" Grid.Row="2"
            d:LayoutOverrides="Width, Height" HorizontalAlignment="Left"
            VerticalAlignment="Center" />
        <TextBlock TextWrapping="Wrap" Text="Price" Grid.Row="3"
            d:LayoutOverrides="Width, Height" HorizontalAlignment="Left"
            VerticalAlignment="Center" />
        <TextBlock TextWrapping="Wrap" Grid.Column="1" d:LayoutOverrides="Height"
            VerticalAlignment="Center" Text="{Binding BookID}" />
        <TextBox Grid.Column="1" Grid.Row="1" TextWrapping="Wrap"
            VerticalAlignment="Center" Text="{Binding Title, Mode=TwoWay}" />
        <TextBox Grid.Column="1" Grid.Row="2" TextWrapping="Wrap"
            VerticalAlignment="Center" Text="{Binding Author, Mode=TwoWay}" />
        <TextBox Grid.Column="1" Grid.Row="3" TextWrapping="Wrap"
            VerticalAlignment="Center"
            Text="{Binding Price, Mode=TwoWay, StringFormat=C}" />
        <Button Content="Remove" Grid.Column="1" HorizontalAlignment="Right"
            Margin="0,-1,0,1" Grid.Row="5" />
    </Grid>
</UserControl>

```

Megjegyzés: A Price mező bekötésekor a Binding StringFormat attribútumát nem a dizájner állította be, sajnos jelen pillanatban erre az Expression Blend 4.0 nem képes. Ezt manuálisan hajtottuk végre.

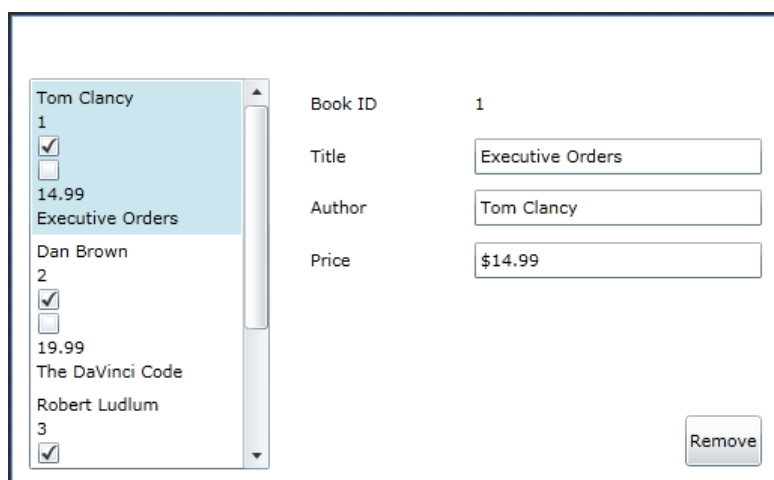
Fordítsunk kiemelt figyelmet az első kiemelt kódrészletre! Ott a **d:DataContext** tulajdonság kerül beállításra. A **d** prefix azt jelzi, hogy ez csak az ún. Design-time DataContext, azaz futásidőben nincs szerepe, csak tervezési időben kerül beállításra az érték. Az eredmény a 10-8 ábrán látható.



10-8 ábra: A BookDetailsView UserControl tervezési időben Design-Time ViewModel bekötése mellett

Az alkalmazás futtatása

Ha visszatérünk a BooksView UserControl szerkesztéséhez, a saját ViewModel-ben implementált „mock” adataink fognak megjelenni mindkét vezérlőben. Futás közben is láthatjuk ezeket az adatokat, ahogyan azt a 10-9 ábra mutatja.



10-9 ábra: Tervezési időben és futási időben is ez a látvány fogad minket

A ViewModel-ek és a VisualStateManager közötti kommunikáció

A vizuális állapotok rendkívül erős eszközt adnak a kezünkbe a felhasználói felületek testreszabásához. Sajnálatos módon, a vizuális állapotok erősen függenek a nézettől. A lazán csatolt ViewModel-ek (vagyis a ViewModel-ből nincs referencia a View-ra) nem tudnak a VisualStateManager-nek közvetlenül üzenni. Persze próbálkozhatunk saját Singleton megoldással, vagy akár messaging-gel is (a lazán csatolt kommunikációról a fejezetben további részleteket találhatunk). Nagy segítséget jelenthetnek a behavior objektumok és a triggerek.

Képzeljük el a következő problémát! Szeretnénk jelezni, ha éppen a ViewModel-ünk a háttérben aszinkron kommunikál. Ezt az állapotot egy **IsLoading** tulajdonság segítségével lehet reprezentálni. Az **IsLoading** tulajdonság megváltozásakor az értéktől függően egy **ProgressBar** példányt kell eltüntetnünk, vagy megjelenítenünk. A legrugalmasabb megoldást akkor készítjük, ha definiálunk két vizuális állapotot, egyet arra az esetre, ha éppen kommunikálunk aszinkron módon, egyet pedig az

alapállapotra. Az **IsLoading** tulajdonság megváltozásának hatására pedig váltunk a két állapot között a megfelelő módon.

A megoldást a triggerek jelentik. Az Expression Blend SDK segítségével képesek vagyunk behavior-öket és triggereket készíteni. Triggert vagy behavior-t tetszőleges vezérlőre el tudunk helyezni. Segítségükkel figyelni tudjuk bizonyos események bekövetkezését. Az pedig ismert, hogy a tulajdonságok setter metódusában egy **PropertyChanged** esemény következhet be. Így nem is lenne bonyolult írni egy olyan trigger-t, amely figyelni annak a vezérlőnek a DataContext-jében található XY tulajdonság **PropertyChanged** esemény által jelzett megváltozását, amely vezérlőre rácsatolták. Az esemény bekövetkezése esetén pedig a paraméterben meghatározott elnevezésű metódust reflection segítségével meghívja. A jó hír, hogy ezt a Behavior-t nekünk már nem kell megírni, az a Silverlight SDK alapsomagjában elérhető, **DataStateBehavior**-nek hívják.

Vezessük be az **IsLoading** tulajdonságot a **BooksViewModel**-be és a **LoadBooks()** híváshoz:

```
private bool isLoading = false;

public bool IsLoading
{
    get { return isLoading; }
    set
    {
        isLoading = value;
        RaisePropertyChanged("IsLoading");
    }
}

// LoadBooks metódus módosítása
public void LoadBooks()
{
    IsLoading = true;
    dataSource.LoadBooks();
}

// LoadBooksCompleted eseménykezelő módosítása
void dataSource_LoadBooksCompleted(object sender,
    Model.ModelEventArgs.LoadBooksCompletedEventArgs e)
{
    if (e.Error != null)
    {
        // Logoljuk és jelenítsünk meg a hibát
    }
    else
    {
        // Töröljük a könyvlistát a memóriából
        Books.Clear();

        // Újra feltöltjük a listát
        foreach (Book book in e.Books)
        {
            BookDetailsViewModel bookDetailsViewModel =
                new BookDetailsViewModel(book, dataSource);
            Books.Add(bookDetailsViewModel);
        }

        SelectedBook = Books.First();
    }
    IsLoading = false;
}
```

A kapcsolódó vizuális állapotok és az animációk:

```
<Grid x:Name="LayoutRoot" Background="White">
  <VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="LoadingStates">
      <VisualStateGroup.Transitions>
        <VisualTransition/>
      </VisualStateGroup.Transitions>
      <VisualState x:Name="Loading">
        <Storyboard>
          <ObjectAnimationUsingKeyFrames
            Storyboard.TargetProperty="(UIElement.Visibility)"
            Storyboard.TargetName="progressBar">
            <DiscreteObjectKeyFrame KeyTime="0">
              <DiscreteObjectKeyFrame.Value>
                <Visibility>Visible</Visibility>
              </DiscreteObjectKeyFrame.Value>
            </DiscreteObjectKeyFrame>
          </ObjectAnimationUsingKeyFrames>
        </Storyboard>
      </VisualState>
      <VisualState x:Name="Loaded"/>
    </VisualStateGroup>
  </VisualStateManager.VisualStateGroups>
```

A **ProgressBar** definíciója és a kapcsolódó **DataStateBehavior**:

```
<ProgressBar x:Name="progressBar" HorizontalAlignment="Left" Height="10"
  VerticalAlignment="Top" Width="100" Margin="11,26,0,0" IsIndeterminate="True"
  Visibility="Collapsed">
  <i:Interaction.Behaviors>
    <ei:DataStateBehavior Binding="{Binding IsLoading}" Value="True"
      TrueState="Loading" FalseState="Loaded"/>
  </i:Interaction.Behaviors>
</ProgressBar>
```

A **DataStateBehavior** DataContext-je maga a **BooksViewModel**, így az **IsLoading** tulajdonsághoz egyszerűen köthetjük. Ha ez az érték **True** lesz, akkor a **Loading** vizuális állapotba kell átnavigálni, ha **False**, akkor a **Loaded** állapotba. Ahogy a vizuális állapotok definíciójánál látjuk, a **ProgressBar** objektum **Visibility** tulajdonságát állítjuk át.

Megjegyzés: Ez a funkció a **Thread.Sleep()** hívás miatt a **MockDataSource** használata esetén nem működik helyesen. Éles adatforrás bekötése esetén, valódi aszinkronitás mellett kifogástalan működést biztosít.

A vizuális állapotokat a dizájnér tetszőlegesen módosíthatja, így az ő lehetősége lesz eldönteni, hogy az adatok töltését hogyan jelezze a felhasználó felé az alkalmazás. Ismét egy lépést tettünk a View és a ViewModel közötti kommunikáció fenntartása mellett a lazán csatolt kapcsolat felé.

Ezzel az MVVM alapú tervezés és fejlesztés alapjainak megismerésén túl is vagyunk. Egy fontos kérdés tisztázása azonban még hátra maradt. Hogyan kötjük be a funkciókat?

Jelen pillanatban az alkalmazás működőképes. Futás közben a kiválasztott elem részletei jelennek meg. Az adatok betöltése automatikusan megtörténik az indulás pillanatában. A következő kérdésekre azonban még keressük a választ:

1. Hogyan töltjük be az adatokat egy gombnyomás hatására?
2. Hogyan töröljük a Remove gombra kattintást követően az elemet az adatforrásból és a listából is egyaránt?

Commanding az MVVM-ben

Lazán csatolt metódushívások

A felhasználói felületen történő, felhasználó által kezdeményezett eseményekre szeretnénk reagálni a ViewModel-ben. Lényegében egy vezérlő által kiváltott esemény bekövetkezte esetén a ViewModel-ben egy metódust kell meghívunk. Fontos megjegyezni, hogy az MVVM tervezési minta meg sem említi a commanding-ot. A *commanding* önmagában is egy tervezési minta, amit jelen esetben az MVVM implementálását segíti elő. Korábbi Silverlight verziókban is fejlesztettek MVVM alapú megoldásokat, annak ellenére, hogy a commanding csak Silverlight 4.0 óta érhető el. A különböző MVVM keretrendszerek és harmadik felek által fejlesztett megoldások néha ún. *attached command*-ok segítségével, néha behavior vagy trigger alapon oldották meg ezt a problémát. A commanding tehát egy lehetőség arra, hogy a felhasználói felületen bekövetkezett események hatására, a ViewModel-ben metódusokat tudjunk meghívni úgy, hogy a nézetnek nem kell ismernie a kapcsolódó ViewModel-t.

MVVM Light Toolkit-ben **RelayCommand**-okat, Prism-ben **DelegateCommand**-okat használunk.

Commanding alkalmazása esetén a ViewModel-ben készíteni kell egy **ICommand** típusú tulajdonságot, amit adatkötni kell. A tulajdonság példányosításakor játszik szerepet a **RelayCommand** objektum. Az alábbi kódrészlet ennek a szerkezetét mutatja be.

```
public class RelayCommand : ICommand
{
    public RelayCommand(Action execute);
    public RelayCommand(Action execute, Func<bool> canExecute);
    public event EventHandler CanExecuteChanged;
    public bool CanExecute(object parameter);
    public void Execute(object parameter);
    public void RaiseCanExecuteChanged();
}
```

A **RelayCommand** két konstruktorral is rendelkezik. Az első konstruktor egy **Action** objektumot vár **execute** néven, ami egy **void** visszatérésű paraméter nélküli metódus. Ez az **Action** arra a metódusra kell, hogy mutasson, amit ennek a command objektumnak meg kell majd hívnia. A második konstruktor rendelkezik még egy **canExecute** elnevezésű **Func<bool>** típusú paraméterrel. A **canExecute** paraméternek olyan metódusra kell mutatnia, ami egy **bool** értékkel tér vissza. Ez a **bool** érték fogja jelezni a command számára, hogy lefuthat-e vagy sem. A command-ok abban az esetben, ha nem futhatnak le, a kapcsolódó vezérlő **IsEnabled** tulajdonságát **false** értékre billentik, vagyis letiltják a vezérlőt. Van aki erre előnyként, és van aki hátrányként tekint. Hátrány lehet abból a szempontból, hogy a dizájnerek lehetőségeit némiképp korlátozzuk, de egyúttal előny is, mert logikailag ez egy helyes automatizmus. A command az **Execute** metódus meghívása esetén a kapcsolódó metódust futtatja le, míg a **CanExecute** hívása esetén a kapcsolódó **CanExecute** logikát. A command-ot ki kell értesíteni arról, hogy ha egy parancs valamilyen okból nem futtatható le, vagy a korábban nem futtatható állapot elhagyhatja. Ezt a manuális lépést a **RaiseCanExecuteChanged** metódus meghívásával lehet végrehajtani. A **RelayCommand**-nak van egy generikus **RelayCommand<T>** párja is. Ezt főleg azokban az esetekben célszerű alkalmazni, amikor a command-on kívül még egy extra értéket, ún. **CommandTarget**-et is szeretnénk átadni. Például a **RemoveBookCommand** meghívása esetén még a **ListBox.SelectedItem**-jét is át szeretnénk adni a **Remove(Book bookToRemove)** metódusnak. Erre többnyire nincs szükség, mert a helyes MVVM implementációkban ezek az információk a ViewModel-ben elérhetők. Néhány esetben azonban hasznos lehet ez a megközelítés is.

A Command Pattern implementálása MVVM-ben

A LoadBooksCommand megvalósítása

A **BooksViewModel** konstruktorában a **LoadBooks()** függvény most automatikusan betölti a termékeket. Ezt inkább egy **Load** gomb megnyomására szeretnénk végrehajtani. Ennek megfelelően készítenünk kell egy **Command** objektumot a **BooksViewModel** osztályban:

```
private RelayCommand loadBooksCommand;

public RelayCommand LoadBooksCommand
{
    get
    {
        if (loadBooksCommand == null)
            loadBooksCommand = new RelayCommand(LoadBooks);
        return loadBooksCommand;
    }
}
```

Ezzel párhuzamosan a **BooksViewModel** konstruktorából el kell távolítani a **LoadBooks()** hívást. Így a könyvek betöltése automatikus nem is történhet meg. A felhasználói felületre egy gomb **Command** tulajdonságába pedig a **LoadBooksCommand** tulajdonságot kell adatkötnünk. Ez történhet Blend-ben a Data panel-ről drag-and-drop-pal, a Blend automatikusan a **Command** tulajdonságot fogja bekötni, de elvégezhetjük a Visual Studio-ból, vagy akár manuálisan is:

```
<Button Content="Load Books" HorizontalAlignment="Left" Height="24" Margin="11,10,0,0"
        VerticalAlignment="Top" Width="158"
        Command="{Binding LoadBooksCommand, Mode=OneWay}"/>
```

A futtatást követően a Load Books gombra kattintva a termékek betöltődnek.

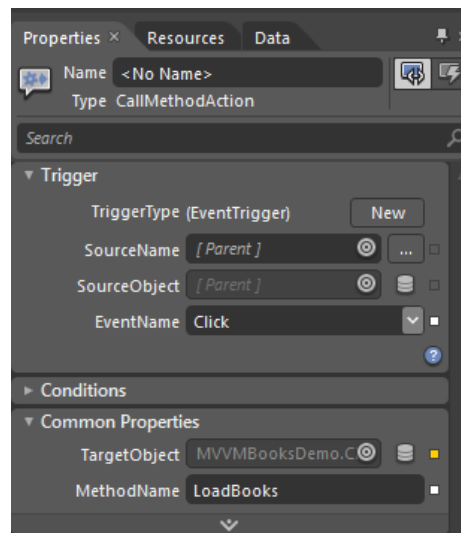
Metódusok bekötése Commanding nélkül

Metódusok hívásához nincs feltétlenül **Command** objektumokra szükség. Egy másik lehetőség a triggerok használata. A **LoadBooks** metódusnak publikusnak kell lennie ebben az esetben, a **Command** tulajdonságra pedig egyáltalán nincs is szükség. Az Expression Blend-ben a Data panelen a tulajdonságok mellett látszanak a metódusok is. Ezeket szintén drag-and-drop-pal be lehet húzni a vezérlőkre. Ennek eredmény az ún. **CallMethodAction** trigger létrejötte:

```
<Button Content="Load Books" HorizontalAlignment="Left" Height="24" Margin="11,10,0,0"
        VerticalAlignment="Top" Width="158">
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="Click">
            <ei:CallMethodAction MethodName="LoadBooks" TargetObject="{Binding}"/>
        </i:EventTrigger>
    </i:Interaction.Triggers>
</Button>
```

Ez a trigger az Expression Blend SDK segítségével készült, így szükség van a **System.Windows.Interactivity.dll**-re is. A trigger jelentése: a **Click** esemény bekövetkeztekor a **LoadBooks** metódus kerül meghívásra. A 10-10 ábrán látható, hogyan lehet a trigger tulajdonságait a Blend-ben beállítani.

Ebben az esetben természetesen **CanExecute** jellegű koncepciók nem vehetők figyelembe. Az ilyen igényeket saját megoldással kell helyettesíteni.



10-10 ábra: A CallMethodAction trigger konfigurációja

A Silverlight-ban tehát több módszert is alkalmazhatunk ViewModel-jeinkben definiált függvényeink meghívására. Mindezt lazán csatolt módon, melynek legfőbb előnye a tesztelhetőség és a dizájnerek szabadsága.

Kommunikáció a ViewModel-ek között

Lazán csatolt kommunikáció

Egy üzleti alkalmazásban a nézetek teljesen függetlenek lehetnek egymástól, akár csak a kapcsolódó ViewModel objektumok. Ennek hatalmas előnye a karbantarthatóság, hiszen ilyen módon a függőségek számát igen alacsonyan tudjuk tartani. Ugyanakkor nem ritka, hogy két olyan ViewModel szeretne információt cserélni, amelyek egymástól „távol”, az alkalmazás különböző részein, külön modulokban vannak, illetve olyan szituáció áll elő, hogy a két ViewModel sem referenciával nem rendelkezik egymásra, és közösen meghivatkozott objektumok sincsenek.

Jelen példában a **RemoveBook()** műveletet szeretnénk megvalósítani. Jól tudjuk, hogy a metódust commanding segítségével könnyen meghívhatjuk a **BookDetailsViewModel**-ben. A metódus törli a könyvet az adatforrásból. Azonban nem csak az adatforrásból kell törölni a könyvet, hanem a **BooksViewModel**-ben található **Books** listából is! Ebben a példában az adatforrás kivált egy eseményt, ha a könyv törlésre került az adatforrásból, ez a **RemoveBookCompleted** esemény. Erre a **BooksViewModel** feliratkozhat, ugyanis ezt az **IBookDataSource** példányt megosztják a ViewModel-ek egymás közt. Hogyan oldanánk meg ezt a problémát, ha nem lenne meg ez a közös adatforrás objektum? Mi lesz akkor például, ha az alkalmazásunkban egy külső modul kíváncsi lesz arra, hogy mi a kiválasztott könyv, mert ez alapján tölt be információkat az Amazon.com-ról?

A megoldás a lazán csatolt kommunikáció. Szükség van egy globális kommunikációs csatornára, amelyen előre megbeszélt, vagy mindenki által ismert üzeneteket küldhetünk. Az üzenetekben érdekelt ViewModel-ek feliratkoznak az ilyen típusú üzenetekre. Ezt a fajta kommunikációt az MVVM Light Toolkit-ben *Messaging*-nek hívják, Prism-ben *EventAggregator*-oknak.

A Messaging a **Messenger** singleton objektum segítségével implementálható. **Register** és **Unregister** metódusai biztosítják a feliratkozást és leiratkozást egy adott típusú üzenet vételére. A **Send** metódus segítségével küldhetünk üzenetet. Az MVVM Light Toolkit használ saját Message objektumokat, mint például a generikus **GenericMessage<T>** osztály. Ezeknek a segítségével könnyen és gyorsan lehet egyszerű üzeneteket küldeni.

Megjegyzés: A szerző véleménye szerint, célszerűbb erősen típusos, konkretizált üzeneteket küldeni, azaz saját specifikus üzenetobjektumokat építeni. Ezáltal elkerülhető a kétértelmű üzenet, illetve a véletlen feliratkozás.

Könyvek törlésének implementálása Messaging segítségével

Üzenetek cseréjére modulok, ViewModellek között közös, mindenki által ismert üzeneteket kell használni. Ezért egy új projekt készül **MVVMBooksDemo.Messaging** néven. Az üzenet a **BookRemovedMessage** nevet kapja.

```
namespace MVVMBooksDemo.Messaging
{
    public class BookRemovedMessage
    {
        public int BookID { get; set; }

        public SelectedBookChangedMessage(int bookID)
        {
            this.BookID = bookID;
        }
    }
}
```

Ezt a projektet (illetve a hozzá tartozó dll-t) minden üzenetet küldeni és fogadni akaró modulnak meg kell hivatkoznia. Az üzenet küldése az MVVM Light Toolkit-ben található **Messenger** osztály **Send** metódusával lehetséges. A **BookDetailsViewModel** osztály **RemoveBook** metódusa a következőképpen változik meg:

```
public void RemoveBook()
{
    dataSource.RemoveBook(BookID);
    Messenger.Default.Send(new BookRemovedMessage(BookID));
}
```

Ezt az üzenetet bárki elolvashatja, aki **BookRemovedMessage** típusú üzenetekre van feliratkozva. A **BooksViewModel** osztály konstruktorában feliratkozhat az ilyen típusú üzenetekre:

```
public BooksViewModel()
{
    if (IsInDesignMode)
    {
        //Design-time-ban design adatokkal dolgozunk
        LoadDesignData();
    }
    else //Egyébként éles adatokkal dolgozunk
    {
        dataSource = new MockBookDataSource();
        dataSource.LoadBooksCompleted +=
            new EventHandler<Model.ModelEventArgs.LoadBooksCompletedEventArgs>(
                dataSource_LoadBooksCompleted);
        // Messaging-et használunk a demonstráció kedvéért
        // dataSource.RemoveBookCompleted +=
        //     new EventHandler<Model.ModelEventArgs.OperationCompletedEventArgs>(
        //         dataSource_RemoveBookCompleted);
    }
}
```

```

    Messenger.Default.Register<BookRemovedMessage>(this, message =>
    {
        int bookID = message.BookID;
        BookDetailsViewModel bookToRemove = Books.Single(p => p.BookID == bookID);
        Books.Remove(bookToRemove);
    });
}
}

```

A **Register** metódus második paraméterében határozzuk meg azt a kódot, amit üzenet beérkezésekor le kell futtatni. A paraméterként kapott **BookID** alapján a **Books** gyűjteményből töröljük a kikeresett **BookDetailsViewModel** példányt.

Futtatáskor a Remove gombra kattintást követően mind az adatforrásból, mind pedig a listából az elem törlésre került.

A két ViewModel közötti kommunikációt lazán csatolva oldottuk meg.

Adatok kezelése MVVM-ben *CollectionViewSource*-szal

A legalapvetőbb kliens oldali adatműveletek közé tartozik a listák rendezése, szűrése, csoportosítása, vagy a navigálás kérdése. Jól tudjuk, hogy a jelenlegi szerkezetünkben is megoldhatók lennének ezek a feladatok. Pusztán a **Books** gyűjtemény elemein kéne rendezést végrehajtanunk, illetve törölnünk belőle elemeket és így tovább. Sajnos ezek jóval összetettebb feladatok, mint elsőre hangzanak. Egy nagyobb n elemű lista rendezése igen sok lépést jelentene Books gyűjteményben. Minden egyes lépést követően kapnánk egy *CollectionChanged* eseményt, ami nem lenne túlságosan hatékony megoldás. A szűrés kérdése még bonyolultabb. A Books listából el kell távolítani elemeket, de új szűrési feltételek meghatározása esetén, a korábban eltávolított elemeket, vissza kell helyezni, mindezt úgy, hogy a rendezéssel is összhangban maradjunk.

Jogosan merül fel bennünk a kérdés, hogy a *DataGrid* akkor miként oldja meg a rendezést automatikusan? Valójában a háttérben némi „varázslat” történik. A *DataGrid* vezérlő a mi esetünkben, nem közvetlenül a Books gyűjteményhez kötne, hanem becsomagolná őt egy **ICollectionView** példányba. Ez a példány gyakorlatilag egy speciális nézet az adatainkra. Anélkül biztosít rendezést, szűrést, csoportosítást, navigációt, hogy az eredeti adatainkat módosítaná.

Ilyen példányt legkönnyebben a **CollectionViewSource** osztály segítségével állíthatunk elő. A **CollectionViewSource** osztály **View** tulajdonsága éppen ilyen **ICollectionView** példányhoz juttat minket.

Az *ICollectionView* bevezetése a *BooksViewModel*-be

Jelen helyzetben a **ListBox** közvetlenül a **Books** tulajdonsághoz köt, ami egy **ObservableCollection<Book>** típusú lista. Sajnos ennek a tulajdonságnak a cseréje **ICollectionView** típusra komoly hátrányt jelentene, és a ViewModel-ünket is át kellene alakítani. A hátrány a tervezési időben történő adatkötésben mutatkozik meg. Sajnos, ha a **Books** tulajdonság egy **ICollectionView**, a Blend többé már nem ismeri fel a benne található típusokat, így elvesztjük a tervezőeszköz támogatását. A legkevésbé fájdalmas megoldás, ha a **Books** tulajdonságot megtartjuk és bevezetünk egy új **BooksView** tulajdonságot az **ICollectionView** számára. Természetesen, ha ehhez kötjük a **ListBox ItemSource** tulajdonságát, ugyanúgy nem lesz tervezésidejű támogatásunk, de bármikor ideiglenesen átkötve a **Books** tulajdonságra (vagy akár csak tervezési módban a **d:DataContext** tulajdonság segítségével) visszaszerezük az tervezésidejű támogatást.

Vegyük fel a **BooksView** tulajdonságot a **BooksViewModel** osztályba:

```
private ICollectionView booksView;

public ICollectionView BooksView
{
    get { return booksView; }
    set
    {
        booksView = value;
        RaisePropertyChanged("BooksView");
    }
}
```

Egészítsük ki a **BooksViewModel** konstruktort:

```
CollectionViewSource cvs = new CollectionViewSource();
cvs.Source = Books;
BooksView = cvs.View;
```

A különböző adat szervezési műveleteket a továbbiakban a **BooksView** tulajdonság segítségével végezzük el. Az elemek felvételét és törlését továbbra is a **Books** gyűjteményben kell elvégezni. A **Books** gyűjteményen bekövetkezett **CollectionChanged** események automatikusan átjönnek a **BooksView** **CollectionChanged** eseményén keresztül.

Rendezés SortDescription-ök segítségével

Az **ICollectionView** interfész rendelkezik egy **SortDescriptions** tulajdonsággal. Ez a lista **SortDescription** típusú. Ha rendezni szeretnénk a listánkat a könyvek címe alapján, akkor a **SortDescriptions** listába kell felvennünk egy új **SortDescription** objektumot, amely konstruktorának a tulajdonság nevét és a lista rendezésének irányát (**ListSortDirection**) kell átadnunk:

```
// Utolsó rendezési irány megőrzése
bool isLastSortAscending = false;

public void SortBooks()
{
    // Ha van már rendezési feltétel meghatározva, akkor töröljük
    if (BooksView.SortDescriptions.Count > 0)
        BooksView.SortDescriptions.Clear();

    // Ha a rendezési irány legutoljára nem növekvő volt
    if (isLastSortAscending)
    {
        BooksView.SortDescriptions.Add(
            new SortDescription("Title", ListSortDirection.Ascending));
    }
    else // Ha a rendezési irány legutoljára nem növekvő volt
    {
        BooksView.SortDescriptions.Add(
            new SortDescription("Title", ListSortDirection.Descending));
    }

    // Legutolsó rendezési irány beállítása
    isLastSortAscending = !isLastSortAscending;
}
```

A **SortBooks()** metódust a commanding szekcióban megismert módon ki kell vezetnünk egy gomb **Click** eseményére. Ezt követően, ha a gombra rákattintunk, a rendezés növekvő sorrendben, ha ismét rákattintunk, csökkenő sorrendben történik.

Szűrés a *Filter delegate* segítségével

Adatok szűrésére az **ICollectionView** típus **Filter** delegate-jének beállításával nyílik lehetőség. A meghatározott függvényt minden listaelemre lefuttatja a **CollectionView**. Ez a függvény egy **object** paramétert fogad a paraméterlistán és **bool** visszatérési értékkel tér vissza. Ha ez az érték **true**, akkor az elem bennmarad a nézetben, ha **false**, akkor nem kerül bele abba.

A **BooksViewModel**-ben a szűrés illesztésének legegyszerűbb módja, ha készítünk egy **Filter** tulajdonságot. A **Filter** tulajdonság hozzáköthető egy **TextBox**-hoz, melyen keresztül könnyedén kereshetünk a könyvek között cím alapján.

A **Filter** tulajdonságot az alábbi módon valósíthatjuk meg a **BooksViewModel** osztályban:

```
private string filter;

public string Filter
{
    get { return filter; }
    set
    {
        filter = value;
        RaisePropertyChanged("Filter");
    }
}
```

A szűrés végrehajtása egy Keresés gombra történő kattintás hatására megy végbe. Ehhez készíteni kell egy **FilterBooks** metódust, amire a command mutathat:

```
public void FilterBooks()
{
    // Ha a szűrő üres, alaphelyzetbe állítjuk a szűrést
    if (string.IsNullOrEmpty(Filter))
    {
        BooksView.Filter = null;
        return;
    }

    // Szűrési feltétel meghatározása
    BooksView.Filter = bookObject =>
    {
        Book currentBook = bookObject as Book;
        // Ha a könyv címe tartalmazza a Filter rész-stringet, akkor kell a könyv
        if (currentBook.Title.Contains(Filter))
            return true;
        else return false;
    };
}
```

Navigáció az adatok között

Az **ICollectionView** típus **CurrentItem** tulajdonsága reprezentálja az aktuális elemet. A **CurrentItem** léptetése a listán előre, illetve hátra az **ICollectionView.MoveNext()** illetve az **ICollectionView.MoveCurrentToPrevious()** metódusok segítségével történhet. Az első illetve az utolsó elemre ugrás az **ICollectionView.MoveCurrentToFirst()** és az **ICollectionView.MoveCurrentToLast()** metódusokkal végezhető el. Ennek megfelelően a **BooksViewModel**-ünkben szükség van egy **MoveToNext()** és egy **MoveToPrevious()** metódusra, amely az előre és a hátra irányú navigációt biztosítja a listán.

A navigációt a ViewModel-ben az alábbi módon valósíthatjuk meg:


```
public void MoveToNext()
{
    // Ugrás a következő elem-re (BooksView.CurrentItem tulajdonság beállítása)
    BooksView.MoveCurrentToNext();

    // Ha ezzel leléptünk a listáról, akkor ugrás az utolsó elemre
    if (BooksView.IsCurrentAfterLast)
        BooksView.MoveCurrentToLast();
}

public void MoveToPrevious()
{
    // Ugrás az előző elem-re (BooksView.CurrentItem tulajdonság beállítása)
    BooksView.MoveCurrentToPrevious();
    // Ha ezzel leléptünk a listáról, akkor ugrás az első elemre
    if (BooksView.IsCurrentBeforeFirst)
        BooksView.MoveCurrentToFirst();
}
```

A **System.Windows.Data** dll-ben szerepel egy **PagedCollectionView** nevű osztály, amely egy másik **ICollectionView** implementáció, használata nagyban megegyezik az eddig leírtakkal. A **PagedCollectionView** azonban a lapozást is támogatja, így a **DataPager** és a **DataForm** vezérlő szívesen dolgozik ezzel az objektummal.

Az **ICollectionView** interfésznek, az interfészt megvalósító osztályoknak és az ezt kihasználó vezérlőknek köszönhetően az eredeti forráslista módosítása, és sok saját kód írása nélkül kaphatunk számos kényelmi funkciót az adatok kezeléséhez.

Adatok érvényességének ellenőrzése

Az elmúlt évek során a fejlesztők sok tapasztalatot és tanulságot szerezhettek meg. Az egyik legfontosabb ezek közül, hogy minden input „az ördögtől való”. Nem tudhatjuk, hogy valóban a megcélzott felhasználó igyekszik az adatokat bevinni az alkalmazáson keresztül, vagy pedig egy rosszindulatú felhasználó („robot”) ül a monitor előtt. Ha az ilyen jellegű kockázatoktól eltekintünk, akkor is élnünk kell azzal a feltételezéssel, hogy a felhasználó elhibázza az adatbevitelt. Például hibás formátumú irányítószámot, adószámot, negatív értékű árat ad meg, vagy éppen egy kötelező mezőt hagy üresen. Mindez azt jelenti, hogy a bemeneteket, az adatok érvényességét **mindig** ellenőrizni kell.

Az ellenőrzési mechanizmus fontos kérdés a Silverlight-ban. A Binding engine-nek köszönhetően a vezérlők és az adatok szinkronizálása automatikusan megtörténik, így nehéz az ellenőrzési logikát közbeilleszteni. Szerencsére a Binding objektum erre a problémára is fel van készítve és számos tulajdonságot biztosít az érvényesség biztosítására.

A **NotifyOnValidationErrors** tulajdonság

Amennyiben ez a Boolean tulajdonság igaz, a **BindingValidationError** esemény kiváltódik az érvényességi hiba bekövetkezésekor és annak megszűnésekor is. Így feliratkozhatunk erre az eseményre és a kezelését végző kódban bármit végrehajthatunk a felhasználói felülettel kapcsolatban. Sajnos ez a megközelítés elég nehézkesen illeszkedik az MVVM gondolatvilágába:

```
<!-- XAML -->
<TextBox Grid.Column="1" Grid.Row="3" TextWrapping="Wrap" VerticalAlignment="Center"
    Text="{Binding Price, Mode=TwoWay, NotifyOnValidationErrors=True, StringFormat=C}"
    BindingValidationError="TextBox_BindingValidationError"/>>
```



```
// C#
private void TextBox_BindingValidationError(object sender, ValidationErrorEventArgs e)
{
    // A ValidationErrorEventArgs alapján eldönthető, hogy validációs hiba megszűnéséről,
    // vagy bekövetkezéséről van szó
}
```

A *ValidatesOnExceptions* tulajdonság

Amennyiben ez a **Boolean** tulajdonság igaz értéket vesz fel, és az adatkötött tulajdonság setter metódusában bármilyen kivétel keletkezik, akkor azt a Binding engine automatikusan érvényességi hibának fogja kezelni. Így az ellenőrzést végző logikát az entitások tulajdonságainak setter metódusaiban lehet elhelyezni. Pozitívum ebben a megközelítésben annak egyszerűsége és az MVVM-mel való összhangja. Ugyanakkor negatívumként említhető, hogy kizárólag a tulajdonságok egyedi ellenőrzése oldható meg ilyen formán, teljes entitás érvényességének és egyéb tulajdonságoktól való függőségek vizsgálata csak korlátozottan, vagy nehezen kivitelezhető.

```
<!-- XAML -->
<TextBox Grid.Column="1" Grid.Row="3" TextWrapping="Wrap" VerticalAlignment="Center"
    Text="{Binding Price, Mode=TwoWay, ValidatesOnExceptions=True, StringFormat=C}"/>

// C#
public decimal Price
{
    get { return book.Price; }
    set
    {
        if (book.Price == value) return;
        if (book.Price < 0) throw new Exception("Price must be positive");
        book.Price = value;
        RaisePropertyChanged("Price");
    }
}
```

A *ValidatesOnDataErrors* tulajdonság

Ez a tulajdonság szorosan kapcsolódik az **IDataErrorInfo** interfész használatához. Ennek az interfésznek a műveleteivel végzi az érvényesség ellenőrzését. A **ValidatesOnExceptions** által biztosított lehetőségekkel ellentétben, ez a mód megengedi a tulajdonságok értékének érvénytelen tartományba állítását is. Az interfész saját tulajdonságok és indexer segítségével jelzi, hogy az adott objektum érvénytelen állapotban található, így összetettebb validációs mechanizmus is megvalósítható a segítségével.

```
<!-- XAML -->
<TextBox Grid.Column="1" Grid.Row="3" TextWrapping="Wrap" VerticalAlignment="Center"
    Text="{Binding Price, Mode=TwoWay, ValidatesOnDataErrors=True, StringFormat=C}"/>

// C#
public class BookDetailsViewModel : ViewModelBase, IDataErrorInfo
{
    ...

    string error = null;
    // Ebben a tulajdonságban tároljuk a hibaüzenetet
    public string Error
    {
        get { return error; }
    }
    // Ez az indexer választja ki a validálandó mezőt
    // Ha a visszatérési érték null, akkor nincs validációs hiba
    // Ha a visszatérési érték string, akkor van validációs, hiba
```

```
// A validációs hibüzenet a visszaadott string
public string this[string columnName]
{
    get
    {
        if (columnName == "Price")
        {
            if (Price < 0)
            {
                error = "Price must be positive";
                return error;
            }
        }
        return null;
    }
}

...
}
```

A *ValidatesOnNotifyDataErrors* tulajdonság

Ennek a tulajdonságnak **True** értékre állításával aszinkron ellenőrzési mechanizmust alakíthatunk ki. A validáció az **INotifyDataErrorInfo** interfész műveleteinek segítségével hajtódik végre. Tipikus példája az aszinkron validációnak, mikor felhasználó nevet kell megadnunk, és miközben már a többi mezőt töltjük ki, a rendszer a háttérben ellenőrzi, hogy az adott felhasználó név foglalt-e. Amennyiben foglalt, azt hibaként fogja értelmezni és aszinkron módon jelzi a problémát a binding engine felé.

Az *INotifyDataErrorInfo* implementálása MVVM-ben

Az **INotifyDataErrorInfo** interfész minden helyzetben alkalmazható az érvényesség ellenőrzésére — függetlenül attól, hogy valóban aszinkron ellenőrzést igényel a tulajdonság, vagy elegendő a szinkron megoldást is. Emiatt érdemes kialakítani egy kollektív validációs mechanizmust. A legegyszerűbb megoldást a **ViewModelBase** osztály kiterjesztése jelenti.

Így készíthetünk egy olyan **ExtendedViewModelBase** osztályt, amely a munka jelentős részét elvégzi:

```
public class ExtendedViewModelBase : ViewModelBase, INotifyDataErrorInfo
{
    //Minden tulajdonsághoz kapcsolódhat több hiba is.
    //Ezeket a hibákat ebben a listában tároljuk
    protected Dictionary<string, List<string>> errorList =
        new Dictionary<string, List<string>>();

    // INotifyDataErrorInfo tag
    public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

    // INotifyDataErrorInfo tag
    // Adott tulajdonsághoz kapcsolódó hibák listájával tér vissza
    public System.Collections.IEnumerable GetErrors(string propertyName)
    {
        CheckErrorCollectionForProperty(propertyName);

        return errorList[propertyName];
    }
    protected void CheckErrorCollectionForProperty(string propertyName)
    {
        // Ha nem volt még a tulajdonsághoz felvéve validációs hiba,
        // akkor inicializálni a kell a listát
        if (!errorList.ContainsKey(propertyName))
        {
            errorList[propertyName] = new List<string>();
        }
    }
}
```

```

// INotifyDataErrorInfo tag
public bool HasErrors
{
    get { return errorList.Values.Count > 0; }
}

// Validációs hiba esetén ez a metódus regisztrálja be a hibát
protected void AddError(string propertyName, string error)
{
    CheckErrorCollectionForProperty(propertyName);
    errorList[propertyName].Add(error);
    RaiseErrorsChanged(propertyName);
}
// Validációs hiba megszűnése esetén ez a metódus törli a validációs hibát
protected void RemoveError(string propertyName, string error)
{
    // Ha megvan a korábban felvett hiba
    if (errorList[propertyName].Contains(error))
    {
        // akkor töröljük
        errorList[propertyName].Remove(error);
        RaiseErrorsChanged(propertyName);
    }
}

protected void RaiseErrorsChanged(string propertyName)
{
    if (ErrorsChanged != null)
    {
        ErrorsChanged(this, new DataErrorsChangedEventArgs(propertyName));
    }
}
}

```

Így a **Price** mező ellenőrzése a **BookDetailsViewModel**-ben a következőképpen történik:

```

public class BookDetailsViewModel : ExtendedViewModelBase
{
    ...
    ...

    public decimal Price
    {
        get { return book.Price; }
        set
        {
            if (book.Price == value) return;
            book.Price = value;
            RaisePropertyChanged("Price");
            ValidatePrice();
        }
    }
    ...

    //Validációs logika
    private void ValidatePrice()
    {
        string priceIsNegativeErrorMessage = "Price must be positive";

        if (book.Price < 0)
        {
            AddError("Price", priceIsNegativeErrorMessage);
        }
        else
        {

```

```
        RemoveError("Price", priceIsNegativeErrorMessage);
    }
}
...
}
```

A **ValidatePrice** metódus hívhatna akár szerver oldali kódot is, amely aszinkron hívást jelentene. A hívás befejezéséhez kapcsolódó **Completed** esemény kezelését végző kódban is hívhatnánk az **AddError** és a **RemoveError** metódusokat. Ezek ugyanis az **ErrorsChanged** eseményt váltják ki az **INotifyDataErrorInfo** interfészen.

Nem maradt más hátra, mint az adatkötésben meghatározni, hogy **INotifyDataErrorInfo** interfész szerint ellenőrizzük az érvényességet:

```
<TextBox Grid.Column="1" Grid.Row="3" TextWrapping="Wrap" VerticalAlignment="Center"
    Text="{Binding Price, Mode=TwoWay, ValidatesOnNotifyDataErrors=True,
    StringFormat=C}"/>
```

Az érvényesség vizsgálata során előforduló hiba megjelenítését a 10-11 ábrán láthatjuk.

Book ID	1
Title	Executive Orders
Author	Tom Clancy
Price	(\$10.00) Price must be positive

10-11 ábra: A Price mező-be -10-es érték beírása után az alapértelmezett validációs mechanizmus

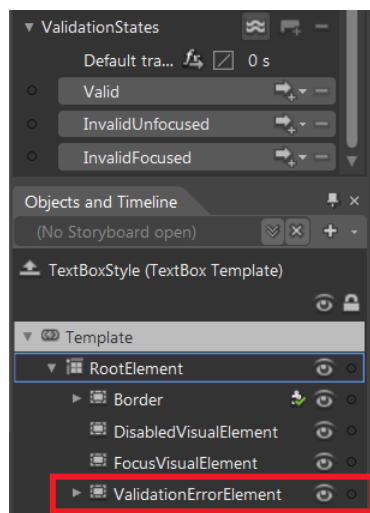
Az ellenőrzéshez kapcsolódó alapértelmezett megjelenés általában a TextBox körüli piros keret és a Tooltip beállításában merül ki. A megjelenés testreszabhatósága fontos igény a megfelelő dizájn kialakítása érdekében, természetesen, ez támogatott.

Érvényességi hibák megjelenítésének testreszabhatósága

A érvényesítést támogató vezérlők **ControlTemplate**-jét jobban megvizsgálva, találunk a vizuális állapotok között egy **ValidationStates** elnevezésű **VisualStateGroup**-ot. Ebbe a csoportba három vizuális állapot tartozik:

1. *Valid* — A vezérlő nem észlelt validációs hibát, az érték érvényes.
2. *InvalidFocused* — A vezérlő érvényességi hibát észlelt, és fókuszban van.
3. *InvalidUnfocuses* — A vezérlő validációs hibát észlelt, de nincs fókuszban.

A vizuális állapotok igény szerint testreszabhatóak. A **TextBox** alapértelmezett vezérlő sablonjában található egy **ValidationErrorElement** elem a vezérlőfában (10-12 ábra). A vizuális állapotok változásai erre a részfára hatnak. Ez az elem szabadon eltávolítható és saját validációs hibamegjelenítés alakítható ki helyette.



10-14 ábra: A TextBox vezérlősablonjának vizuális állapotai a validációs mechanizmus számára, továbbá az állapotok által módosított ValidationErrorMessageElement vizuális fa részlete

A fenti technikák segítségével a validációs mechanizmust és a kapcsolódó megjelenítést könnyen testreszabhatjuk igényeinknek megfelelően.

A ViewModel-ek Unit tesztelése

A unit tesztek írása rendkívül fontos része a jó minőségű szoftverek fejlesztésének. Segítségükkel komponenseink helyes viselkedését tesztelhetjük különféle körülmények között. Így egyetlen funkcióhoz nem csak egy, hanem több teszt is tartozhat. Például, a **RemoveBooks** metódus hívásánál normális körülménynek tekinthető, ha olyan könyvet próbálunk törölni, ami létezik, így ennek a műveletnek kifogástalanul kell működnie. Természetesen arra is tesztelni kell a funkciót, ha olyan könyvet próbálunk törölni, ami nem létezik. Első gondolatunk alapján, ilyen az alkalmazásunkban nem fordulhat elő, de bármikor véthetünk olyan hibát, ami ilyen körülményekhez vezet. Ezekben az esetben is a **RemoveBooks** metódusnak helyes kell kezelnie a bemenetet. Unit tesztet természetesen nem minden egyes metódushoz kell készíteni, csak a komponensek fontosabb műveleteit kell megtesztelni, amelyek a komponenssel való interakciót biztosítják a külvilág, az egyéb komponensek számára.

A unit tesztek írásának fontosságát mi sem bizonyítja jobban, hogy egy komplett megközelítési mód is épült köréje. Ez az ún. *Test-Driven Development* (TDD), azaz a tesztvezérelt fejlesztés. Ebben a gondolkodásmódban a fejlesztők először a teszteseteket készítik el, majd ezt követően írják meg úgy a lényegi kódot, hogy az a teszteseteket kielégítse.

A ViewModel-ek bevezetésének egyik fő oka éppen a tesztelhetőség — a helyes funkcionális működés ellenőrizhetőségének — megteremtése volt. Bármilyen módosítás esetén, a korábban megírt tesztek újrafuttathatók, ezáltal ellenőrizhetővé válik, hogy a módosításaink után az alkalmazás funkciói még mindig helyesen működnek-e.

A Silverlight alkalmazások unit tesztelése rendkívül fontos. Sajnos, a Silverlight sajátos run-time modelljének köszönhetően a Visual Studio saját unit teszt keretrendszere nem használható Silverlight alkalmazások tesztelésére. A Silverlight Toolkit-ben azonban elérhető a Silverlight Unit Testing Framework. Ez a Framework egy önálló Silverlight alkalmazást készít, saját felülettel, melyet ugyanúgy hosztolnunk kell, mint a tényleges alkalmazást.

Unit teszt készítése a BooksViewModel-hez

A **BooksViewModel**-ben a **LoadBooks()** metódus helyes működését tesztelni kell. A **LoadBooks()** akkor működik helyesen, ha a hívást követően a **Books** gyűjtemény feltöltésre kerül.

Unit teszt készítéséhez létre kell hoznunk egy új, Silverlight Unit Test Application típusú projektet. Esetünkben ennek a projektnek a neve **MVVMBooksDemo.Client.UnitTests** lesz és ezt az projektet is

ugyanaz a site fogja hosztolni, mint az alkalmazásunkat. A projekt felvételét követően egy új ASPX és egy új HTML oldal is létrejön az **MVVMBooksDemo.Web** projektben. Ezek a unit teszt alkalmazást hosztoló oldalak.

A ViewModel-lek tesztelésének egyik kulcsfontosságú problémája az aszinkronitás tesztelése. A **LoadBooks()** metódus egy aszinkron hívás, visszatérés nélkül. Nem tudni, mikor fejeződik be a futása — és ennek megfelelően, mikor tér vissza — a unit tesztet ezzel a szemléletmóddal kell elkészíteni. Ebben számos metódus segíti munkánkat, melyek a **SilverlightTest** ösosztályból származnak. Néhány ezek közül a teljesség igénye nélkül:

- **EnqueueCallback(Action testCallbackDelegate)** — Ez a metódus a paraméterben átadott kódrészletet (metódust) fogja meghívni aszinkron módon. Lényegében egy teszhívási sorba helyezi el a metódust, ahol majd meghívásra kerül.
- **EnqueueConditional(Func<bool> conditionalDelegate)** — A teszt hívási sort várakoztatja addig, amíg a delegate **true** értékkel nem tér vissza
- **EnqueueDelay(double miliseconds)** — A megadott ideig várakoztatja a teszt hívási sort, mielőtt tovább engedné.
- **EnqueueTestComplete()** — Jelzi a teszt hívási sornak, hogy a teszt véget ért.

Ennek megfelelően az alábbi unit tesztet írhatjuk:

```
[TestClass]
public class Tests : SilverlightTest
{
    [TestMethod]
    [Asynchronous] // Aszinkron unit teszt lesz
    public void LoadBooksTest()
    {
        // Ez a flag jelzi, ha az aszinkron művelet végetért
        bool isAsyncOperationCompleted = false;

        // Mock adatforrást használunk
        IBookDataSource dataSource = new MockBookDataSource();

        // A BooksViewModel-t teszteljük
        BooksViewModel viewModel = new BooksViewModel(dataSource);

        // Ha a könyvet betöltődtek, a flag-et állítsuk be.
        dataSource.LoadBooksCompleted += (s,e) => isAsyncOperationCompleted = true;

        // Hívjuk meg a LoadBooks metódust a ViewModel-ben
        EnqueueCallback(viewModel.LoadBooks);

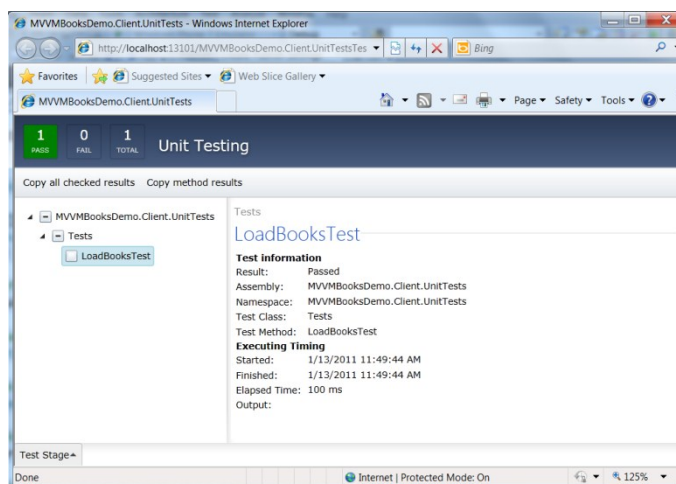
        // Várjunk míg a flag beállításra kerül
        EnqueueConditional(() => isAsyncOperationCompleted);

        // Ha továbbléptünk, ellenőrizzük a feltételt
        EnqueueCallback(() => Assert.IsTrue(viewModel.Books.Count > 0,
            "Books count is zero!"));

        // Jelezni kell, hogy a teszt ezzel véget ért
        EnqueueTestComplete();
    }
}
```

Az **EnqueueCallback** metódussal meghívjuk a **LoadBooks** metódust, majd az **EnqueueConditional** metódussal várakoztatjuk a hívási sort addig, amíg az **isAsyncOperationCompleted** flag értéke **true** nem lesz. A **dataSource** miután betöltötte a könyveket, egy **LoadBooksCompleted** esemény emel, amire feliratkozunk, és ez állítja **true** értékre az flag-et. A hívási sor következő eleme az **Assert.IsTrue** hívás, ezzel ellenőrizzük az eredményt. Ha a **Books** lista elemszáma nagyobb, mint nulla, akkor sikerrel jártunk, egyébként jelezni kell a hibát. A **EnqueueTestComplete** metódussal jelezzük, hogy a teszt véget ért. A

unit teszt alkalmazást hosztoló oldal futtatásával indíthatók a unit tesztek. A tesztek egy saját keretalkalmazásban futnak le, ahogyan azt a 10-15 ábra mutatja.



10-15 ábra: A LoadBooksTest unit teszt futtatásának eredménye

Összefoglalás

A Mode-View-ViewModel architektúrális mintát a prezentációs rétegben célszerű bevezetni. Ennek segítségével alkalmazásunk felhasználói felülete tesztelhető, karbantartható lesz, és jelentősen egyszerűsíti a fejlesztő és dizájnér közötti együttműködést.

Ebben a megközelítési módban több elem is segíti a hatékony projektmunkát:

- a ViewModel-ek koncepciója,
- a View-val való laza integráció
- a ViewModel-ben exponált tulajdonságok, és parancsok, melyeket Command pattern, illetve Behavior-ök segítségével elérhetővé tettünk
- illetve az Expression Blend és a ViewModel-jeinkbe beépített tervezésidejű támogatás.

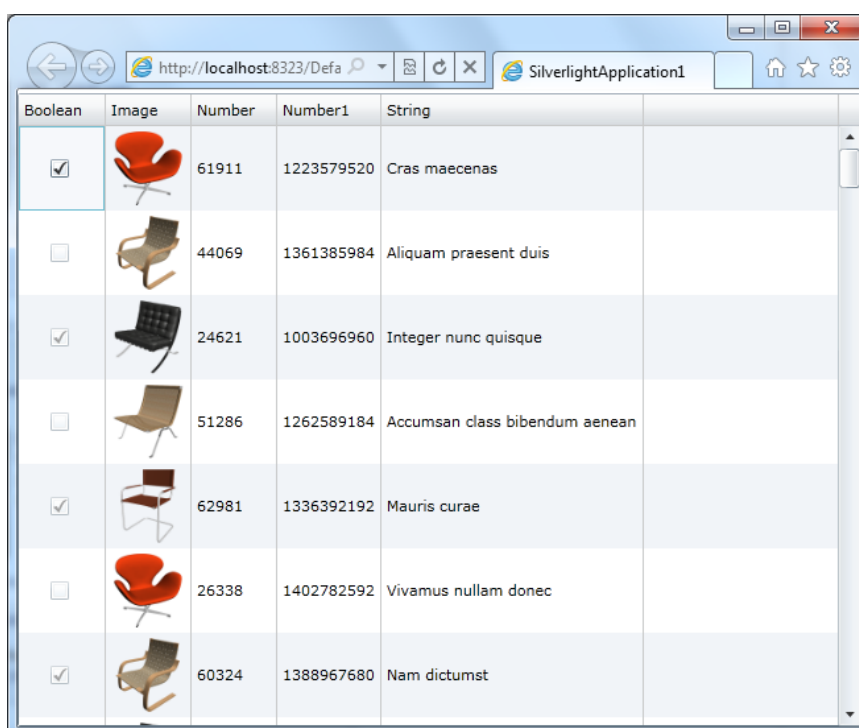
A ViewModel-jeink felépítésének köszönhetően alkalmazásunk a Silverlight UnitTesting Framework segítségével egyszerűen tesztelhető, akár aszinkron műveletekre is. A ViewModel-be épített ellenőrzési mechanizmusok, illetve a haladó adatkezelést támogató **ICollectionView** példányok jelentősen leegyszerűsítik az adatokhoz kapcsolódó feladatok elvégzését és bonyolultabb vezérlők támogatását.

Az MVVM alapú megoldások megvalósításakor mindig szembekerülhetünk kisebb problémákkal, melyekre megoldást kell találnunk azért, hogy megfelelő absztrakciós szintet alkalmazzunk, a komponensek között laza csatoltságot biztosítsunk, és alacsonyan tartsuk a függőségeket. Ezen lehetőségek jelentős részét az MVVM keretrendszerek, vagy éppen behavior-ök, triggerek biztosítják számunkra, néhány esetben azonban saját megoldásokat kell majd kialakítanunk. A fejezet legfontosabb üzeneteként elmondható, hogy professzionális üzleti alkalmazások fejlesztése során érdemes a Model-View-ViewModel minta használatával dolgozni.

11. Összetett adatok megjelenítése és kezelése

Táblázatos adatmegjelenítés

Amikor üzleti alkalmazásokat fejlesztünk Silverlight technológia segítségével, olyan elemeket vegyítünk a pehelysúlyú keretrendszer innovatív megoldásaival, amelyek már sok-sok éve megszokottak az üzleti alkalmazások világában. A megszokás, illetve a jó felhasználhatóság miatt nehezen szabadulhatunk meg tőlük. Ilyen például a táblázatos megjelenítési forma is, amelyet szinte az összes .NET keretrendszerre épülő technológia megvalósít. Ezek közül a Silverlight sem kivétel, amely a 2.0-ás verzió óta rendelkezik ezzel a vezérlőtípussal, és az minden egyes újabb kiadásban továbbfejlődik. A **DataGrid** vezérlőt jól használhatjuk saját megoldásokhoz is, azonban igazi erejének kihasználására a RIA Services képes.



Boolean	Image	Number	Number1	String
☒		61911	1223579520	Cras maecenas
☐		44069	1361385984	Aliquam praesent dui
☒		24621	1003696960	Integer nunc quisque
☐		51286	1262589184	Accumsan class bibendum aenean
☒		62981	1336392192	Mauris curae
☐		26338	1402782592	Vivamus nullam donec
☒		60324	1388967680	Nam dictumst

11-1 ábra: Táblázatos adatok megjelenítése Silverlight alatt

A DataGrid vezérlő alapvető használata

A **DataGrid** vezérlő segítségével rugalmasan tudunk táblázatos formában megjeleníteni összefüggő adathalmazokat. A vezérlő alapvetően szöveges és **CheckBox** típusú oszlopokat támogat, azonban sablonok segítségével ezt kedvünkre átalakíthatjuk és kibővíthetjük. A megjelenítésen túl számos beépített eszközt tartalmaz, amelyek az adatok szerkesztését, érvényességének ellenőrzését és szűrését segíti. Továbbá — ahogy azt a Silverlight világában már megszokhattuk — a megjelenítés is teljes mértékben testreszabható.

A fejezet további részeiben a Visual Studio 2010-ben fogunk dolgozni. Hozzunk létre egy egyszerű Silverlight 4 alkalmazást a hozzá tartozó web alkalmazással együtt! Hozzunk létre egy termékeket tároló entitás osztályt, amely négy tulajdonsággal és egy felparaméterezett konstruktorral rendelkezik:

```
public class Product
{
    public Product( string name, DateTime releaseDate, double price, bool? isAvailable )
    {
        this.Name = name;
        this.ReleaseDate = releaseDate;
        this.Price = price;
        this.IsAvailable = isAvailable;
    }

    public string Name { get; set; }
    public DateTime ReleaseDate { get; set; }
    public double Price { get; set; }
    public bool? IsAvailable { get; set; }
}
```

Van már egy entitáosztályunk, amelyet fel tudunk használni. Mielőtt létrehoznánk a mintaadatokat, magát a felhasználói felületet is létre kell hoznunk. El kell helyeznünk egy **DataGrid** vezérlőt a felületen, ám ehhez néhány további lépésre van szükségünk. Két út közül választhatunk (a piros és a kék kapszula, ezt már jól tudjuk ☺).

- A Visual Studio szerkesztője tud nekünk segíteni, ha a Toolboxról húzzuk fel a felületre a vezérlőt, akkor ő automatikusan felveszi a névtér hivatkozásokat és az assembly referenciákat, tovább nekünk nem kell törődni vele.
- Választhatjuk a hosszabb utat is, amikor mi vesszük fel a szükséges komponensek hivatkozásait. Először is szükségünk lesz a **System.Windows.Controls.Data.dll** referenciájára, ezt adjuk a projekthez. Utána minden egyes XAML kódban, ahol a vezérlőt használni szeretnénk, szükségünk lesz a következő névtér hivatkozásra is:

```
xmlns:sdk="http://schemas.microsoft.com/winfx/2006/xaml/presentation/sdk"
```

Vigyázzunk, hogy a fenti névtér hivatkozás kizárólag Silverlight 4 alatt működik, Silverlight 3 esetében magára az assemblyre és a CLR névtérre kell rámutatnunk! Ha ezzel készen vagyunk, akkor a következő kód segítségével hozzunk létre egy üres **DataGrid** vezérlőt:

```
<Grid x:Name="LayoutRoot" Background="White">
    <sdk:DataGrid x:Name="ProductDG"
        AutoGenerateColumns="True" />
</Grid>
```

A **ProductDG** objektumot a **Loaded** eseményvezérlőn belül fogjuk adatokkal feltölteni. Az adatkollekció eltárolásához **ObservableCollection**-t használunk, amely specifikusan Silverlight és WPF adatkötési feladatokhoz készült. Ugyanúgy enumerálható, mint a **List<T>**, azonban nagy előnye, hogy segíteni tud a tárolt entitások változáskövetésében.

A **DataGrid** rendelkezik egy **ItemsSource** tulajdonsággal, amelyen keresztül a valódi adatkötés történik meg. Silverlight és WPF technológiák esetében egy rendkívül fejlett adatkötő motor dolgozik a háttérben, amelynek az esetek nagy részében csupán arra van szüksége, hogy meghatározzuk, milyen elemet és mihez szeretnénk kötni, a további feladatokat automatikusan elvégzi. Miután átadtuk a kollekciónkat, a motor látja, hogy a memóriában egy listába szervezett entitáshalmaz helyezkedik el, és a vezérlőnek ezeket az objektumpéldányokat fogja átadni. Ha az objektumpéldány olyan típusú tulajdonságokkal rendelkezik, amelyeket az adott vezérlő ismer, akkor implicit módon meg is tudja azt jeleníteni a Silverlight, ha nem, akkor nekünk kell a vezérlőhöz egy saját sablont készítenünk. A következő példában pár elemet helyezünk el a listában, és a listát közvetlenül a **DataGrid** vezérlőhöz kötjük. A 11-2 ábra mutatja be a működő alkalmazást.

```

public MainPage()
{
    InitializeComponent();

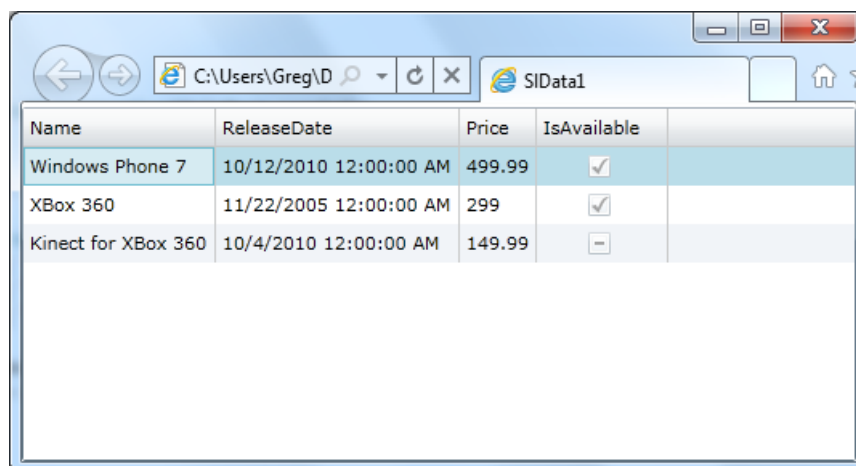
    this.Loaded += new RoutedEventHandler(MainPage_Loaded);
}

void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    ObservableCollection<Product> items = new ObservableCollection<Product>();

    items.Add( new Product( "Windows Phone 7",
                           new DateTime( 2010, 10, 12 ), 499.99, true ) );
    items.Add( new Product( "XBox 360",
                           new DateTime( 2005, 11, 22 ), 299.0, true ) );
    items.Add( new Product( "Kinect for XBox 360",
                           new DateTime( 2010, 10, 4 ), 149.99, null ) );

    ProductDG.ItemsSource = items;
}

```



Name	ReleaseDate	Price	IsAvailable
Windows Phone 7	10/12/2010 12:00:00 AM	499.99	☒
XBox 360	11/22/2005 12:00:00 AM	299	☒
Kinect for XBox 360	10/4/2010 12:00:00 AM	149.99	☐

11-2 ábra: DataGrid automatikusan generált oszlopokkal

Saját oszlopsablonok létrehozása

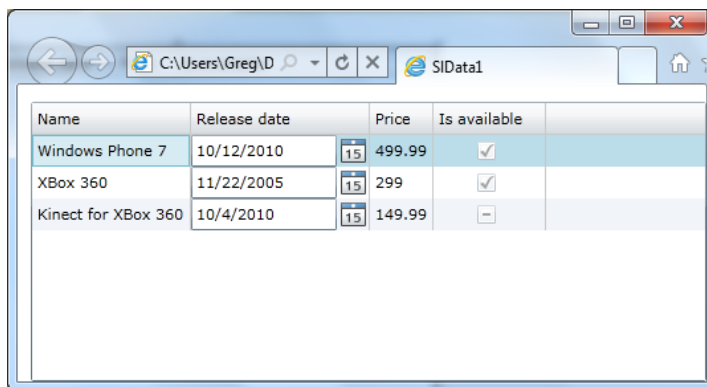
Az előző példában láthattuk, hogy az **AutoGenerateColumns** tulajdonság **True** értékűre állításával a **DataGrid** vezérlő automatikusan el tudta készíteni az oszlopokat, és a hozzájuk tartozó adatokat megjeleníteni. A **string**, **double** és **bool?** típusokkal nincs is semmilyen probléma, és első pillantásra a **DateTime** típussal sem. Azonban ez annak tudható be, hogy a **DateTime** is rendelkezik **ToString()** metódussal, amelyet az adatkötő motor fel tudott használni. Ha szeretnénk, hogy ne szöveggént jelenjen meg a dátum, hanem egy külön vezérlő segítségével, akkor saját oszlopsablonokat kell létrehoznunk. Ha mégis az automatikus generálásnál szeretnénk maradni (nagy oszlopszám esetén érdemes), akkor a generálás folyamatát mi is befolyásolni tudjuk az **AutoGeneratingColumn** esemény segítségével, ahol a megfelelő helyen belenyúlhatunk a folyamatba. A jelenlegi példában azonban az összes oszlopot explicit módon fogjuk definiálni. A **DataGrid.Columns** gyűjteményében tudjuk ezeket a sablonokat elhelyezni és adat kötni. Alapvetően három oszloptípusunk van:

- A **DataGridTextColumn**, amely egy **TextBox**-ot jelenít meg az adatok reprezentálásához;
- a **DataGridCheckBoxColumn**, amely egy **CheckBox**-hoz köti az adatokat;
- a **DataGridTemplateColumn** egy univerzális típus, amelyen belül tetszőleges vezérlőt helyezhetünk el egy **DataTemplate**-en belül.

Az összes oszlopnak a **Header** tulajdonságon keresztül tudunk címkét adni, míg a **Binding** tulajdonságon keresztül vagyunk képesek adatot kötni azokhoz.

Az alábbi kód az előzővel megegyező példát ad, kiegészítve a dátum oszlopot, ahol egy **DatePicker** segítségével jelenítjük meg az adott dátumot. A 11-3 ábra a futtatás eredményét mutatja be:

```
<Grid x:Name="LayoutRoot" Background="White">
    <sdk:DataGrid x:Name="ProductDG"
        AutoGenerateColumns="False"
        Margin="10,10,0,0">
        <sdk:DataGrid.Columns>
            <sdk:DataGridTextColumn Header="Name" Binding="{Binding Name}" />
            <sdk:DataGridTemplateColumn Header="Release date">
                <sdk:DataGridTemplateColumn.CellTemplate>
                    <DataTemplate>
                        <sdk:DatePicker SelectedDate="{Binding ReleaseDate}" />
                    </DataTemplate>
                </sdk:DataGridTemplateColumn.CellTemplate>
            </sdk:DataGridTemplateColumn>
            <sdk:DataGridTextColumn Header="Price" Binding="{Binding Price}" />
            <sdk:DataGridCheckBoxColumn Header="Is available" Binding="{Binding
IsAvailable}" />
        </sdk:DataGrid.Columns>
    </sdk:DataGrid>
</Grid>
```



The screenshot shows a window titled 'SIData1' with a DataGrid containing the following data:

Name	Release date	Price	Is available
Windows Phone 7	10/12/2010	499.99	☒
XBox 360	11/22/2005	299	☒
Kinect for Xbox 360	10/4/2010	149.99	☐

11-3 ábra: DataGrid saját dátum oszloppal

Adatok csoportosítása, rendezése és szűrése

A **DataGrid** egy rendkívül fejlett vezérlő, amely lehetőséget biztosít számunkra különböző szűrési opciók alkalmazására. Ehhez szükségünk van egy **PagedCollectionView** objektumpéldányra, amely bármely **IEnumerable** alapú kollekciót képes csoportosíthatóvá, rendezhetővé és szűrhetővé tenni. Gondoljunk úgy erre az osztályra, mint egy rétegre a **DataGrid** vezérlő és az adatkötés forrása között. Képes konverziókat elvégezni oda-vissza egy szabályrendszer alapján, és ezt így nem kell nekünk megtennünk. Csoportosításhoz a **GroupDescriptions** gyűjteményen fogunk egy új szabályt alkalmazni, amely egy **PropertyGroupDescription** lesz. Segítségével egy adott tulajdonság neve mentén csoportosíthatjuk az adatokat.

A rendezéshez egy **SortDescription** objektumpéldányra van szükségünk, amely szintén egy tulajdonsághoz kapcsolódva végzi el a műveletet, azonban itt már azt is meg kell adnunk, hogy a rendezés **ListSortDirection.Ascending** — növekvő — vagy **ListSortDirection.Descending** — csökkenő-e.

Szabadon definiálhatunk szűrési feltételeket is. Ehhez a **Filter** predikátumot kell elkészítenünk, amely a paraméterlistán egy **T** típusú objektumot kap, és ezzel az objektummal tudunk dolgozni. Visszatérési értéként egy olyan logikai értéket kell visszaadnunk, amely alapján a Silverlight el tudja dönteni, hogy az

adott elem megjeleníthető-e. Az alábbi kód mind a három lehetőséget bemutatja, a futtatás eredménye pedig a 11-4 ábrán látható:

```
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    ObservableCollection<Product> items = new ObservableCollection<Product>();
    items.Add( new Product( "Windows Phone 7", new DateTime( 2010, 10, 12 ),
        499.99, true ) );
    items.Add( new Product( "XBox 360", new DateTime( 2005, 11, 22 ), 299.0, true ) );
    items.Add( new Product( "Kinect for Xbox 360", new DateTime( 2010, 10, 4 ),
        149.99, null ) );

    PagedCollectionView view = new PagedCollectionView( items );
    view.GroupDescriptions.Add( new PropertyGroupDescription( "Name" ) );
    view.SortDescriptions.Add( new SortDescription( "ReleaseDate",
        ListSortDirection.Ascending ) );
    view.Filter = new Predicate<object>( ( p ) =>
    {
        return ((Product)p).IsAvailable.HasValue;
    } );
    ProductDG.ItemsSource = view;
}
```

Name	ReleaseDate	Price	IsAvailable
Name: XBox 360 (1 item)			
XBox 360	11/22/2005 12:00:00 AM	299	☒
Name: Windows Phone 7 (1 item)			
Windows Phone 7	10/12/2010 12:00:00 AM	499.99	☒

11-4 ábra: DataGrid csoportosítással, szűréssel és rendezéssel

Dialógusformába rendezett adatok megjelenítése

A Silverlight 3-ban megjelent egy új vezérlő (**DataForm**), amely dialógus formátumú adatok megjelenítésére és kezelésére használható. Az eddigiek során megismert adatkötési módszert használva ez a vezérlő képes automatikusan legenerálni számunkra a felületet, a tulajdonságok típusait szem előtt tartva, és az entitást leíró szerkezet alapján az érvényességet meghatározó szabályokat is képes alkalmazni.

Adatbeviteli felület létrehozása a DataForm vezérlővel

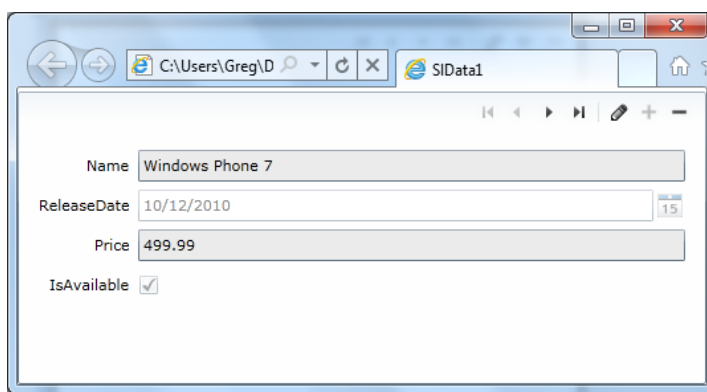
Mivel a Silverlight fizikai méretét a Microsoft a lehető legkisebbre próbálja optimalizálni, ezért vannak olyan vezérlők, melyek nem érhetők el a Silverlight SDK részeként. Ezeket egy külön csomagban, nyílt forráskóddal teszi közzé a Microsoft az alábbi címen: <http://silverlight.codeplex.com/>

Ennek a csomagnak része a **DataForm** vezérlő is. Használata szinte semmiben sem különbözik a **DataGrid** vezérlőétől, kivéve a kódpéldában látható **CommandButtonsVisibility** tulajdonságot, amely azért felel, hogy a **DataForm** mely kezelőszervei jelenjenek meg a felületen. Értékei a következők lehetnek (egy vagy több is használható egyszerre): **All**, **Edit**, **Add**, **Commit**, **Cancel**, **Navigation**, **Delete**. A következő példában egy egyszerű példán keresztül nézzük meg a vezérlő használatát, illetve a 11-5 ábra bemutatja a futó alkalmazást:

```
<!-- XAML kódrészlet -->
<toolkit:DataForm x:Name="ProductDF"
                  AutoEdit="False"
                  CommandButtonsVisibility="All" />

// --- C# kód
void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    ObservableCollection<Product> items = new ObservableCollection<Product>();
    items.Add( new Product( "Windows Phone 7", new DateTime( 2010, 10, 12 ), 499.99,
        true ) );
    items.Add( new Product( "XBox 360", new DateTime( 2005, 11, 22 ), 299.0, true ) );
    items.Add( new Product( "Kinect for XBox 360", new DateTime( 2010, 10, 4 ),
        149.99, null ) );

    ProductDF.ItemsSource = items;
}
```



11-5 ábra: Objektumkollekció megjelenítése DataForm segítségével

A 11-5 ábrán látható, hogy a hozzáadás gomb nincs engedélyezve. Ez annak tudható be, hogy ha bővíthetővé szeretnénk tenni egy kollekciót, akkor az adatkötés forrásának meg kell valósítania az **ICollectionView** vagy az **IEditableCollectionView** interfészek valamelyikét, ez pedig a fenti kódban nem történik meg.

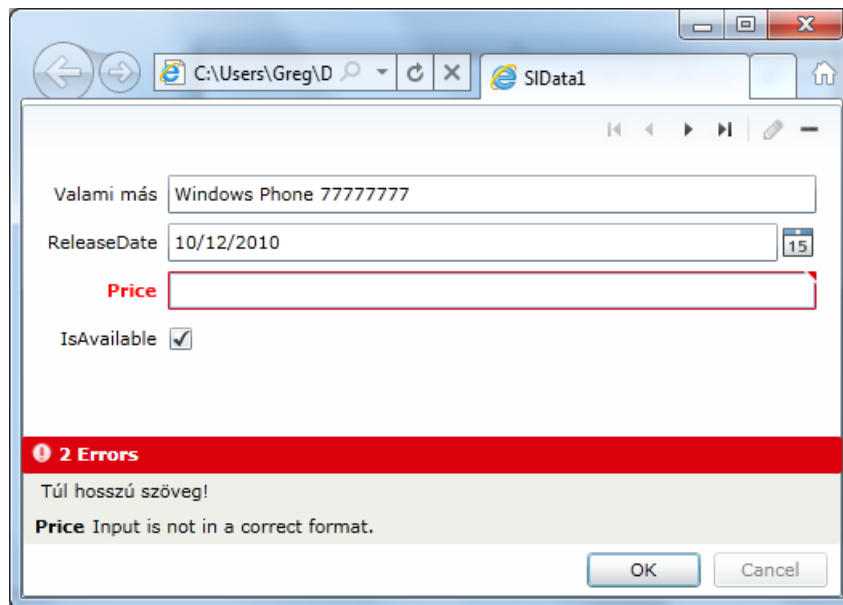
Az entitások érvényességét leíró szabályokat is megadhatunk attribútumok segítségével, ezeket az entitás osztály tulajdonságaihoz kell kötnünk. Egy-egy mezőt általában saját névvel (címkével) szeretnénk megjeleníteni. Mivel a **DataForm** automatikusan generálja le a felületet az adatforrásnak megfelelően, és ott az osztályban szereplő tulajdonság nevét használja, ezért a **Display** attribútum segítségével saját nevet adhatunk neki. Fontos lehet adatbázisba visszaírt szöveges adatok esetében korlátozni egy string hosszát is, erre a **StringLength** attribútum nyújt segítséget. Szintén adatbázisok során lehet fontos, hogy bizonyos mezők kitöltését kötelezővé tegyük. Ezt a **Required** attribútum biztosítja, amelynek egy hibaüzenetet is megadhatunk, amelyet a mező tartalmának üresen hagyásakor jelenít meg. A gyakran használtak a lenti példában szerepelnek, a teljes lista pedig itt található meg: <http://bit.ly/gQmvt>. A 11-6 ábra az alábbi kódhoz tartozó futó alkalmazást mutatja be:

```
[StringLength( 10, ErrorMessage = "Túl hosszú szöveg!" )]
[Display( Name = "Valami más" )]
public string Name { get; set; }

public DateTime ReleaseDate { get; set; }

[Required( ErrorMessage = "Kötelező megadni ezt a mezőt!!" )]
public double Price { get; set; }

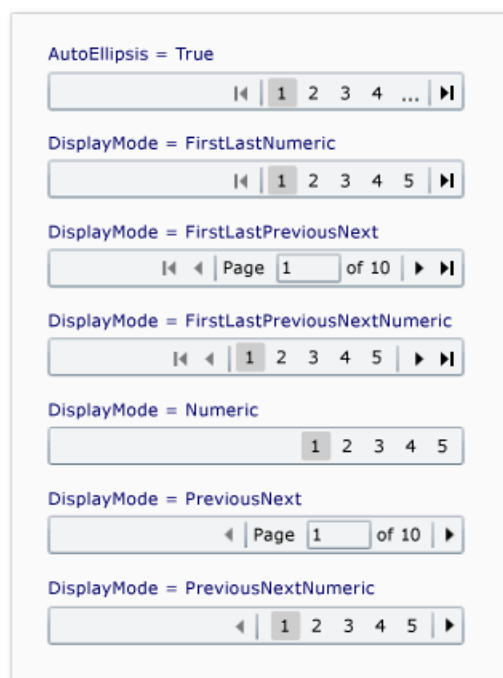
public bool? IsAvailable { get; set; }
```



11-6 ábra: Adatok érvényességének ellenőrzése a DataForm vezérlőben

Lapozás az adatok között

Gondoljunk csak arra, mi történik, ha több száz, ezer vagy százezer entitás szerepel egy gyűjteményben, és mi ezeket szeretnénk megjeleníteni egy **DataGrid** vagy más vezérlő segítségével? Az alkalmazás használhatatlanul lassú lesz, nem beszélve arról, hogy teljes mértékben áttekinthetetlen. Erre jelent megoldást a lapozás, amely adatainkat kisebb halmazokra bontja, és mi egyszerre csak egy kisebb részhalmazt látunk a felületen. Már láttunk ilyen gyűjteményt a **PagedCollectionView** bemutatásakor. Most is ezt fogjuk használni egy **DataPager** vezérlő segítségével, amelyet könnyedén elhelyezhetünk alkalmazásunkban, és az többféle módon is megkönnyítheti számunkra az adatok közötti gyors navigálást. Fontos vigyáznunk arra, hogy a vezérlőt pontosan ahhoz az adatforráshoz kössük hozzá, amelyhez a **DataGrid** — vagy más vezérlő — is kötve van. A 11-7 ábra a vezérlő különböző megjelenési módjait mutatja be.



11-7 ábra: A DataPager megjelenítési módjai

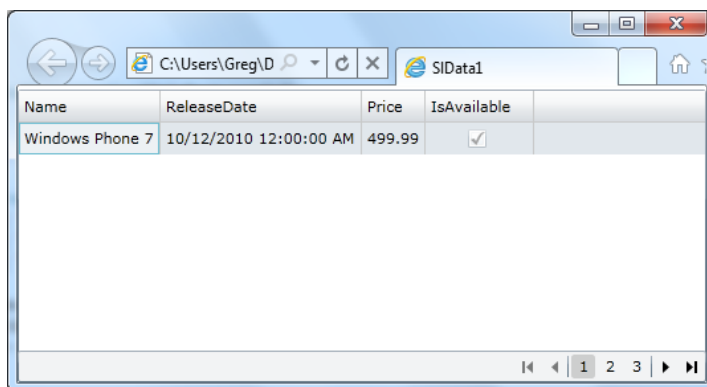
Fontos még beállítani, hogy hány adatot szeretnénk látni egy lapon. Ezt a számot érdemes 100-nál kisebb értékre felvenni, hogy jól áttekinthető megjelenést kapjunk. Az alábbi kód hatását a 11-8 ábra mutatja be:

```
<!-- XAML kód -->

<Grid x:Name="LayoutRoot" Background="White">
    <sdk:DataGrid x:Name="ProductDG"
        AutoGenerateColumns="True" />
    <sdk:DataPager x:Name="ProductPager"
        PageSize="1"
        DisplayMode="FirstLastPreviousNextNumeric"/>
</Grid>

// --- C# kód

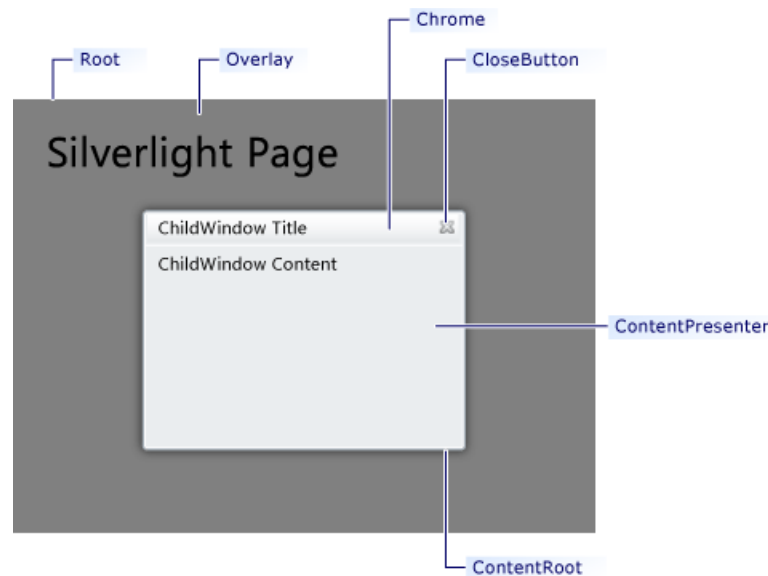
PagedCollectionView view = new PagedCollectionView( items.ToList() );
ProductPager.Source = view;
ProductDG.ItemsSource = view;
```



11-8 ábra: *DataPager* segítségével lapozható adatok megjelenítése

Virtuális ablakok a Silverlightban

A Silverlight 3-ban megjelent **ChildWindow** vezérlő segítségével virtuális ablakokat hozhatunk létre, amelyek segítségével különböző felhasználói interakciókat tudunk lebonyolítani. Ezek az ablakok vizuális elemekkel kiemelik a tartalmat a felhasználói felület többi részéből, ezzel az új tartalomra irányítva a felhasználó figyelmét. Miért nem **MessageBox**-ot használunk? A **ChildWindow** sokkal nagyobb rugalmasságot biztosít, a megjelenítését teljes mértékben átalakíthatjuk, tartalma pedig tetszőlegesen változtatható, hiszen az nem csupán szöveges információkat képes megjeleníteni. A 11-9 ábrán a **ChildWindow** vezérlő felépítését tekinthetjük meg.



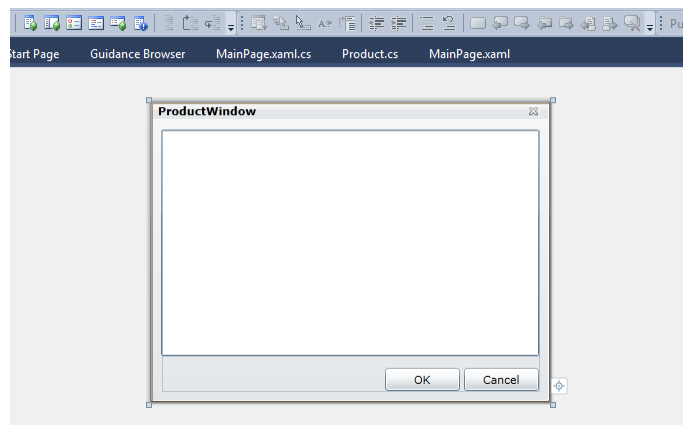
11-9 ábra: A ChildWindow vezérlő felépítése

ChildWindow létrehozása és adatkötése

A Visual Studióhoz készült Silverlight Tools feltelepítése után egy új sablon segítségével hozhatunk létre egy **ChildWindow** vezérlőt. A sablon alapértelmezett felületet biztosít, így azt nem szükséges manuálisan leprogramoznunk. A következő lépések segítségével egy tetszőleges Silverlight 4 projektben hozzunk létre egy új virtuális ablakot:

1. Solution Explorerben válasszuk ki a Silverlight projektet!
2. Jobb kattintással válasszuk az **Add New Item**-et.
3. A **Category** panelen válasszuk ki a **Silverlight** kategóriát!
4. Itt válasszuk a **Silverlight Child Window** sablont és adjunk neki tetszőleges nevet!

A tervezőfelületen megjelenő **ChildWindow** vezérlő a 11-10 ábrán látható.



11-10 ábra: ChildWindow vezérlő a Visual Studio tervezőfelületén

Adjuk a vezérlőnek a **ProductWindow** nevet, utalva arra, hogy a példában elkészített **Product** gyűjteményt jelenítjük rajta meg. Ahhoz, hogy ezt megtegyük, adatkötést kell majd használnunk, mivel nincs olyan hivatkozási forma, hogy **ProductWindow.ProductDataGrid** vagy valami hasonló. A **ProductWindow DataContext** tulajdonságán keresztül tudjuk átadni a Products példányokat tartalmazó kollekciót, és a vezérlőn belül ezt kell felhasználnunk. A **ChildWindow** felületén elhelyezünk egy **DataGrid** vezérlőt, amelynek az **ItemsSource** tulajdonságát az üres „**{Binding}**” kifejezés segítségével adat kötik. Azért ezt a kifejezést használjuk, mivel nem tudunk a konténerre névvel hivatkozni. Az üres kötéssel azt mondjuk meg a Silverlightnak, hogy mi egy magasabb szinten található adatforráshoz szeretnénk kötni,

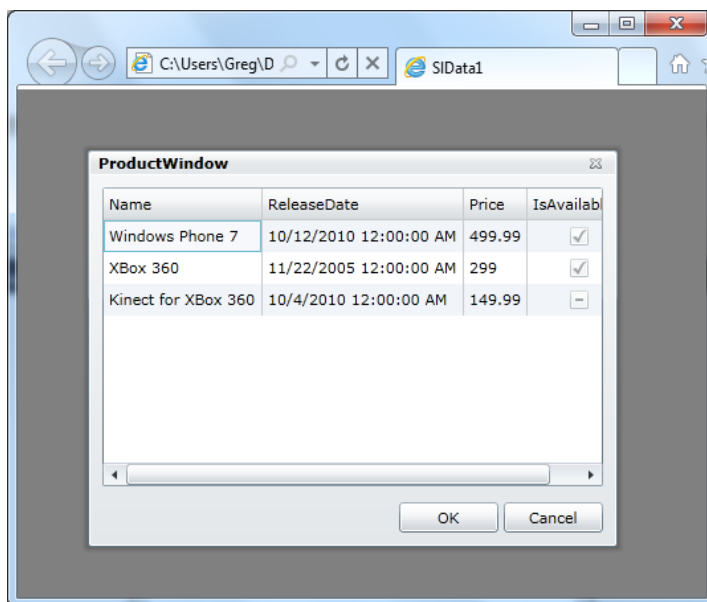
jelen esetben ez maga az egész **ChildWindow**. Miután az adatkötéssel is végeztünk, a **Show()** metódus segítségével megjelenítjük az ablakot. Amíg a virtuális ablak aktív, addig a felhasználói felület többi része nem hozzáférhető. Az alábbi kód a saját készítésű **ProductWindow** használatát mutatja be, a 11-11 ábrán pedig a működő alkalmazás látható:

```
<!-- ProductWindow.xaml -->
...
<sdk:DataGrid x:Name="ProductDG"
    Grid.Row="0"
    ItemsSource="{Binding}"
    AutoGenerateColumns="True" />
...

// --- MainPage.xaml.cs

void MainPage_Loaded(object sender, RoutedEventArgs e)
{
    ObservableCollection<Product> items = new ObservableCollection<Product>();
    items.Add( new Product( "Windows Phone 7", new DateTime( 2010, 10, 12 ),
        499.99, true ) );
    items.Add( new Product( "XBox 360", new DateTime( 2005, 11, 22 ), 299.0, true ) );
    items.Add( new Product( "Kinect for Xbox 360", new DateTime( 2010, 10, 4 ),
        149.99, null ) );

    ProductWindow window = new ProductWindow();
    window.DataContext = items;
    window.Show();
}
```



11-11. ábra: Saját **ChildWindow**, amely adatokat jelenít meg

Hibaüzenetek megjelenítése **ChildWindow** segítségével

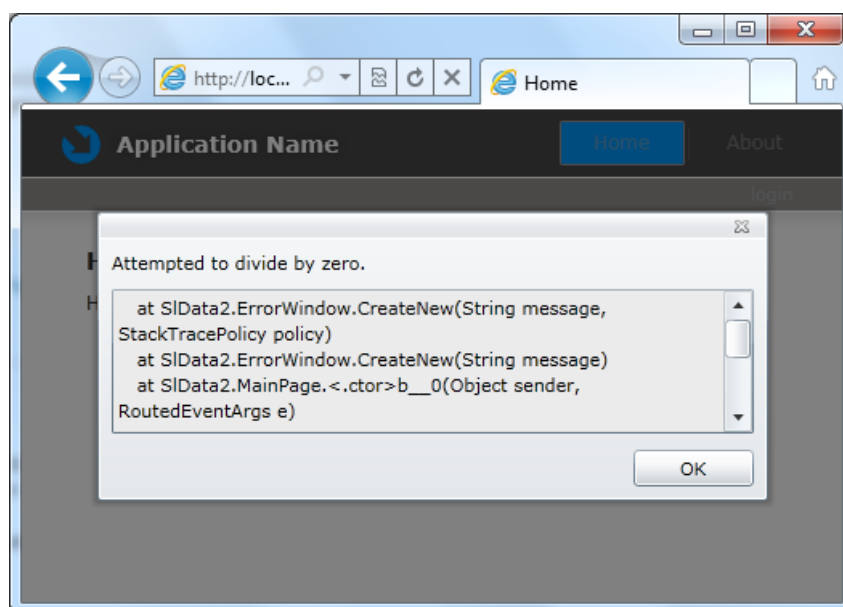
A Silverlight 4 Business Application projektsablon beépítve tartalmazza a **ChildWindow** egy speciális változatát, amellyel tetszőleges üzenetet, illetve kivételek által generált üzeneteket tudunk megjeleníteni. Ahhoz hogy kipróbáljuk, hozzunk létre egy ilyen alkalmazást Visual Studióban, majd a **MainPage.xaml.cs** konstruktorában helyezzünk el az alábbi kódot:

```

this.Loaded += ( sender, e ) =>
{
    try
    {
        int zero = 0;
        int n = 1 / zero;
    }
    catch ( Exception ex )
    {
        ErrorWindow.CreateNew( ex.Message );
    }
};

```

Az 11-12 ábra a futó alkalmazást mutatja be.



11-12 ábra: *ErrorWindow* a Silverlight 4-es sablonban

A Silverlight Toolkit

Vezérlők, melyek nem a Silverlight SDK részei

Ahogy azt már korábban említettem, a Microsoft arra törekszik, hogy a Silverlight komponensek fizikai méretét alacsonyan tartsa, ennek következményeképp nem tudtak mindent a keretrendszer alapvető részévé tenni, amit a fejlesztői társadalom hiányolt. Ennek orvoslására született meg a Silverlight Toolkit, amely már külön kiadásként Windows Phone 7-re is elérhető, és olyan vezérlőknek a teljes arzenálját tartalmazza, amelyeknek kifejlesztése saját kezűleg nagy munka lenne, és ugyanakkor bizonyos feladatokhoz pótolhatatlanok. A csomag teljes mértékben nyílt forráskódú, így ha bármit át szeretnénk alakítani, akkor megvan rá a lehetőségünk.

A Toolkit az alábbi címről letölthető, illetve a weblapon működő futtatható mintákat is találunk:

<http://silverlight.codeplex.com/>

Összefoglalás

Ebben a fejezetben megismertük a Silverlight üzleti alkalmazások fejlesztéséhez szükséges vezérlőit. Ezek többsége elengedhetetlen LOB alkalmazások megvalósításához. Természetesen, a fejezet példái csak nagyon egyszerű, izolált példák voltak, de azok segítenek a vezérlők működésének megértésében, azok felhasználásában.

12. Moduláris alkalmazások fejlesztése

Lazán csatolt rendszerek

Architektúrális megfontolások

A Model-View-ViewModel architektúrális mintával foglalkozó fejezetben hangsúlyoztuk, hogy a karbantartható, tesztelhető kód egyik legfontosabb alapkövetelménye a megfelelően dekomponált rendszer. Minél nagyobb méretű egy rendszer, annál nagyobb hangsúlyt kap, hogy az architektúrát és ezzel együtt az alkalmazást önálló kisebb egységekre daraboljuk. Az alkalmazás modulokba történő felosztásának számos előnye van:

- *Karbantartható kód* — Minden modulnak jól elkülöníthető feladata van, amiért felelős. Módosítások, hibakeresés esetén pontosan tudjuk, hova kell nyúlni.
- *Könnyebb tesztelhetőség* — Az önálló modulok tesztelése jóval egyszerűbb, hiszen minden modul csak egy adott funkcionalitás-halmazért felelős. Igaz, a modulok közötti kommunikáció és együttműködés vizsgálata — így az integrációs tesztelés is — fontos szerepet tölt majd be.
- *Fizikai elkülöníthetőség* — Nagyméretű alkalmazások esetén a különböző modulokon különböző fejlesztők, különböző csapatok dolgozhatnak.

Ugyanakkor az architektúra ilyen jellegű kialakítása, felosztása egyáltalán nem egyszerű feladat. Ez az igény a teljes architektúrára érvényes, így nem meglepő módon a prezentációs rétegre is. Az ebben a rétegben szerepet játszó Silverlight alkalmazás felhasználói felületét és a kapcsolódó alkalmazásokat is feloszthatjuk különálló nézetekre, modulokra. Az így készült alkalmazásokat kompozit alkalmazásoknak nevezzük.

A modularizációhoz hasonló másik nagyon fontos terület a plug-in rendszerek támogatása. Miért ne készíthetnénk olyan alkalmazást, amely előszeretettel alkalmaz később, az alkalmazás megszületése után készült kiegészítéseket, plug-in-eket? Ilyen típusú alkalmazások tervezésekor szem előtt kell tartanunk, hogy a különböző kiegészítések később készülnek el, azokat mégis úgy kell tudnunk az alkalmazásunkba integrálni, mintha már a kezdetek óta ott lennének. Ilyen tekintetben hasonlít a koncepció a moduláris alkalmazások készítéséhez. Pusztán az a különbség, hogy míg a moduláris tervezésnél, bár a modulok helyettesíthetőnek készülnek, alapvetően megtervezzük és implementáljuk őket, csak éppen lazán csatoltan kerülnek bekötésre, addig a plug-in rendszereknél, a plug-in-ek általában jóval később, egy megfelelő keretrendszer segítségével kerülnek dinamikusan becsatolásra.

A modulok bevezetésével könnyebben karbantartható kódállományt kapunk, ezáltal a függőségeket drasztikus mértékben csökkenthetjük az alkalmazásunkban.

Függőségek kezelése — Dependency Injection

Tegyük fel, hogy van egy adatforrás modulunk, melynek az a szerepe, hogy kérésre előállítson bizonyos adatokat. Ezt a modult számos másik modul használja, mindegyik pontosan ugyanúgy. Éppen ezért a modul működését és interakciós felületét egy interfészben definiáljuk. Ameddig a modulok mindegyikének közvetlenül kell hivatkoznia az adatforrás modorra, addig mindegyik modul erősen függ ettől és ennek implementációjától. A modularizáció akkor lehet igazán előnyös, ha a modulok ettől az adatforrás modultól közvetlenül nem függnek, csupán annyit várnának el, hogy rendelkezésre álljon egy olyan objektum, amely az adatforrás egy jól definiált interfészét megvalósítja.

A tényleges függés így csak futás közben alakulhat ki. Ezt az elvet — vagyis a függőségek futásidejű feloldását — nevezzük *dependency injection*-nek (DI).

A dependency injection koncepciója a .NET-ben akár Reflection, akár DI keretrendszerek segítségével megoldható. Az alábbi felsorolás néhány jól használható DI keretrendszert mutat be:

- *Microsoft Unity* — Eredetileg az Enterprise Library alkalmazás blokkjai közül az egyik, DI keretrendszer, amelyet a *Microsoft Patterns and Practices* csapata fejlesztett ki. A könyv írásának pillanatában a Unity 2.0-ás verziója a legfrissebb. A Prism 2.1-es változatáig modulok lazán csatolására kizárólag a Unityt használta, de a Prism 4.0-val ez megoldható a *Managed Extensibility Framework* segítségével is. Ugyanakkor a Unity támogatása DI keretrendszerként továbbra is megmaradt.
- *Ninject* — A Nate Kohari által készített nyílt forráskódú DI keretrendszer, amely támogatja a Silverlight integrációt is.
- *Caliburn* — Egyre növekvő népszerűségnek örvendő, nyílt forráskódú, kliens oldali keretrendszer WPF és Silverlight számára, amely főként az MVVM architektúráis minta támogatását tűzte ki célul, s mint ilyen, a dependency injectiont is fontos elemnek tartja és támogatja.
- *Managed Extensibility Framework* — A Managed Extensibility Framework (MEF) valójában nem igazi DI keretrendszer. Eredeti célkitűzése, hogy könnyen építhessünk plug-in-okat támogató alkalmazásokat. A Visual Studio 2010 számos kiegészítését is a MEF segítségével lehet elkészíteni. Népszerűsége azonban a .NET 4.0-ás és a Silverlight 4.0-ás jelenlétnek és a könnyű, hatékony használatának köszönhetően drasztikus mértékben megnőtt olyan helyzetekben is, ahol eredetileg DI keretrendszereket alkalmaztak, így például a Prism 4.0-ás változatában is használhatjuk a MEF-et Unity helyett.

A Managed Extensibility Framework

A Managed Extensibility Framework (MEF) segítségével könnyen építhetünk olyan alkalmazásokat, amelyek támogatják kiegészítések (plug-in-ek) dinamikus becsatolását.

MEF-fel dolgozni olyan érzés, mint a LEGO darabkákat összeilleszteni és végül felépíteni egy várat. A MEF használata 3 fő lépésből áll:

1. Exportálás
2. Importálás
3. Komponálás

Nézzük meg ezeket a lépéseket közelebbről!

Exportálás

Az első lépés a kiegészítések elkészítése. Szükség van egy modulra, plug-in-re vagy bármilyen objektumra, amely alkalmas arra, hogy becsatolják. Jeleznünk kell a Managed Extensibility Framework számára, hogy egy komponens készen áll arra, hogy becsatolják — vagyis az általa kínált szolgáltatásokat más modulok felhasználják. Ezt a lépést hívja a MEF terminológia *exportálás*nak.

Az alábbi kódrészleten látható **SimpleDataSource** objektumot szeretnénk exportálni. MEF-ben ezeket az objektumokat (komponenseket), amelyek szolgáltatásokat kínálnak fel, illetve a felkínált szolgáltatásokat fogyasztják, *part*-nak (részegység) hívják. Azt, hogy egy objektum szolgáltatást kínál fel a külvilág számára, az **[Export]** attribútum segítségével jelezheti. Az **[Export]** attribútum a **System.ComponentModel.Composition** névtérben (a hasonló nevű assemblyben) érhető el.

```

namespace MEFDemo
{
    [Export]
    public class SimpleDataSource
    {
        public SimpleDataSource()
        {
            //Inicializálás
            //...
        }
        public IEnumerable GetData()
        {
            //Adatok betöltése
            return ...;
        }
    }
}

```

Importálás

A második lépésben az alkalmazásban ki kell jelölni azokat a csatolási pontokat, ahová a kiegészítéseket be lehet tölteni. Ezt a lépést a MEF terminológia *importálás*nak nevezi. Jelen esetben meg kell határoznunk, hogy a **SimpleDataSource** példányt hova szeretnénk betölteni. A komponens (part) számára készíteni kell egy tulajdonságot, ahová az beimportálható. Az importálási szándékot az **[Import]** attribútum jelzi. Az alábbi kódrészleten látható az importálás megvalósítása:

```

namespace MEFDemo
{
    [Export]
    public partial class MainDataView : UserControl
    {
        [Import]
        public SimpleDataSource DataSource { get; set; }

        public MainDataView()
        {
            InitializeComponent();
        }
    }
}

```

Jelen esetben az importálás helye egy másik komponens (part), melyet szintén exportálásra szánunk. A **DataSource** változóba kell a MEF-nek egy **SimpleDataSource** példányt létrehoznia. Bár ennek a mechanizmusnak a segítségével komplex export-import struktúrák is felépíthetők, igyekezzünk könnyen követhető és érthető megoldásokat építeni!

Komponálás

A harmadik és egyben utolsó lépés a komponálás volt. Ebben a fázisban jelezzük a MEF-nek, hogy végezze el a kompozíció összeállítását, vagyis a felkínált (**Export**) és keresett (**Import**) szolgáltatások összeillesztését. Az utasítást a legegyszerűbben a **CompositionInitializer** objektum **SatisfyImports(object attributedPart)** metódusával végezhetjük el. A **SatisfyImport()** metódus a paraméterlistán egy objektumot vár, melyben vannak olyan tulajdonságok, ahol az **Import** attribútum megtalálható. A MEF az összes betöltött típust végignézi és betölti az **Export** attribútummal ellátott objektumokat. Ha talál megfelelő **Export**-ot, akkor azt hozzárendeli az **Import**-hoz. Ha a betöltött exportban szintén található olyan tulajdonság, ami rendelkezik az **Import** attribútummal, akkor az ahhoz illeszkedő exportokat is igyekszik becsatolni és így tovább. Ha a **SatisfyImports()** metódus paraméterben olyan objektumot kap, amiben nincs **Import** attribútum egyetlen tulajdonságon sem,

akkor az exportok betöltése bár megtörténik, de egyéb helyeken található import kérések nem kerülnek kielégítésre.

Az alábbi kódrészleten láthatjuk a kompozíció előállítására vonatkozó kérést és a betöltött part felhasználását.

```
namespace MEFDemo
{
    public partial class MainPage : UserControl
    {
        [Import]
        public MainDataView DataView { get; set; }

        public MainPage()
        {
            InitializeComponent();

            // Kompozíció összeállítása
            CompositionInitializer.SatisfyImports(this);
        }
    }
}
```

A **SatisfyImports()** hívást követően a **DataView** tulajdonság értéke egy **MainDataView** példány, melyben a **DataSource** tulajdonság értéke egy **SampleDataSource** példány. A MEF elvégzi számunkra a példányosítást és a példányok összekapcsolását.

Az importálás lehetőségei

Az exportálás mechanizmusa természetesen ennél összetettebb lehet és a finomhangolásra is mutatkozik bőven igény. Így többek között olyan kérdésekre kell válaszokat keresnünk, mint:

- Milyen típusú objektumok illeszthetők egy importhoz pontosan?
- Mi történik, ha egyetlen import helyére több export is beilleszthető?
- Ha egyetlen típusú exportot több helyre is be kell illeszteni, akkor mindenhova új példány vagy ugyanaz a példány kerül becsatolásra?

Importálás interfészek alapján

A korábbi példákban az importálás mindig az adott export típusa alapján történt. Amennyiben az **Export** attribútumnak nem mondjuk meg, hogy milyen típusként szeretnénk exportálni az adott részegységet, akkor a részegység típusát fogja automatikusan használni az illesztéshez. Ugyanez igaz az importálás oldalára is. Amennyiben nem határozzuk meg az **Import** attribútumnak, hogy milyen típusú elemeket lehet az adott tulajdonsághoz illeszteni, a tulajdonság típusát fogja közvetlenül használni.

A függőségek csökkentése érdekében, a kellő absztrakciós szint bevezetéséhez a különböző partokat többnyire egy őstípuson keresztül szokás exportálni, illetve importálni. Ez lehet egy tetszőleges ősosztály, ugyanakkor praktikus lehet, ha ez az őstípus egy interfész. Ezzel a megközelítéssel ugyanis a plug-in-ek számára egy közös felületet biztosítunk, azaz meghatározhatjuk, hogy az alkalmazásunk hogyan tud majd kommunikálni a később betöltött plug-in-ekkel.

Az előző példából kiindulva, a **SimpleDataSource** osztály exportálása az **IDataSource** interfészen keresztül történhet, melyet az osztály implementál. Az **Export** attribútum első paraméterében az exportálandó típusinformációt kell átadni. Ezt a **typeof** operátor segítségével szerezzük meg.

Az **IDataSource** interfész forráskódja az alábbi:


```
namespace MEFDemo
{
    public interface IDataSource
    {
        IEnumerable GetData();
    }
}
```

A **SampleDataSource** exportálását az **IDataSource** típus segítségével végezzük:

```
namespace MEFDemo
{
    [Export(typeof(IDataSource))]
    public class SimpleDataSource : IDataSource
    {
        public SimpleDataSource()
        {
            //Inicializálás
            //...
        }
        public IEnumerable GetData()
        {
            //Adatok betöltése
            return ...;
        }
    }
}
```

Ugyanez a megközelítés a **MainDataView** exportálására és importálására is átvihető. Az őstípus lehet saját típus, egy interfész, de akár a **UserControl** őstípus is:

```
namespace MEFDemo
{
    [Export(typeof(UserControl))]
    public partial class MainDataView : UserControl
    {
        [Import]
        public IDataSource DataSource { get; set; }

        public MainDataView()
        {
            InitializeComponent();
        }
    }
}
```

A fenti kódrészlet eredményképp a **MainDataView** importálása is történhet **UserControl** típusként. Ezzel egy időben a **DataSource** tulajdonság típusát megváltoztattuk **IDataSource** típusra, hiszen így szeretnénk interakcióba lépni az ide csatolt adatforrás példánnyal, akármilyen típusú is legyen az. Az **Import** attribútum paramétereként meghatározható explicit módon, hogy a csatoláskor **IDataSource** típusokat lehet illeszteni, de szükségtelen, ugyanis az alapértelmezett viselkedésnek megfelelően, a MEF a tulajdonság típusa alapján próbál exportot keresni, ez pedig egyébként is az **IDataSource** típus.

Importálás string contractok alapján

Nemcsak típusok alapján lehet exportálni, hanem ún. string contractok alapján is. Ez esetben exportáláskor egy string segítségével nevet adunk az exportnak. Importáláskor pedig e név szerint keressük a megfelelő export definíciót, amint az alábbi kód példa is mutatja:

```
// Exportálás

namespace MEFDemo
{
    [Export("SampleData")]
    public class SimpleDataSource : IDataSource
    {
        public SimpleDataSource()
        {
            // Inicializálás
            //...
        }
        public IEnumerable GetData()
        {
            // Adatok betöltése
            return ...;
        }
    }
}

// Importálás

namespace MEFDemo
{
    [Export(typeof(UserControl))]
    public partial class MainDataView : UserControl
    {
        [Import("SampleData")]
        public IDataSource DataSource { get; set; }

        public MainDataView()
        {
            InitializeComponent();
        }
    }
}
```

Az *ImportingConstructor* használata

Az eddigiekben elfogadtuk a tényt, hogy a MEF-től kapunk egy példányt, amikor kell. Hogy jön létre ez a példány? A MEF alapértelmezett viselkedésének megfelelően az alapértelmezett konstruktort (az objektum paraméterek nélküli konstruktort) fogja használni a példány elkészítésére. Számos esetben előfordulhat, hogy ez a viselkedés számunkra nem kielégítő. Saját magunk szeretnénk meghatározni, hogy melyik paraméteres konstruktor kerüljön meghívásra. Gondoljunk csak a tesztelhető ViewModelekre, ahol például az adatforrás objektum paraméterként átadható, így teszteléskor könnyedén cserélhető valamilyen „mock” adatforrásra.

Ilyen esetekben jöhet jól az **ImportingConstructor** attribútum. Ezt az attribútumot a kívánt konstruktor fölé illesztve meghatározzuk a MEF számára, hogy melyik konstruktort kell használnia példányosításkor. A paraméter lista egyes elemei pedig szintén MEF segítségével importálhatóak. Az alábbi kódrészlet egy **ViewModel** objektum exportálását mutatja be, ahol az importáláskor egy speciális paraméteres konstruktort használunk:

```
namespace MEFDemo.ViewModels
{
    [Export]
    public class MainDataViewModel
    {
        IDataSource dataSource;
```

```

    [ImportingConstructor]
    public MainDataViewModel([Import]IDataSource dataSource)
    {
        this.dataSource = dataSource;
    }
}

```

Megjegyzés: A konstruktor paraméterét nem kötelező az **Import** attribútummal jelölni, a MEF automatikusan megpróbálja kielégíteni a függőséget érvényes export alapján.

Az **MainDataView** vezérlőhöz tartozó **ViewModel** bekötése az alábbi kóddal történik:

```

namespace MEFDemo
{
    [Export(typeof(UserControl))]
    public partial class MainDataView : UserControl
    {
        [Import]
        public MainDataViewModel ViewModel { get; set; }
        public MainDataView()
        {
            InitializeComponent();
        }
    }
}

```

A **MainDataViewModel** példányosításakor nyilvánvalóan szükség lesz egy **IDataSource** példányra, amit a MEF automatikusan előállít és a **MainDataViewModel** konstruktorának átad. Ezzel nekünk már nem kell foglalkoznunk.

CreationPolicy használata példányok megosztására

Eddig azzal a feltételezéssel éltünk, hogy a MEF mindig készített nekünk egy példányt az adott objektumból, amikor szükségünk volt rá. Sok esetben az adott objektumokból nem célszerű több különböző példányt készítenünk és külön kezelni őket, hanem csupán egyetlen példányt szeretnénk megosztani az importált részek között. Ez a viselkedésforma az **Import** attribútum **RequiredCreationPolicy** nevű paramétere segítségével befolyásolható. A paraméter értéke egy **CreationPolicy** felsorolt típus kiválasztott értéke:

- **Any** — Ez az alapértelmezett viselkedés. Amíg a részegység vagy az importáló nem kéri explicit módon a **NonShared** módot, addig a működés a **Shared** állapotnak megfelelő.
- **Shared** — Egyetlen példány készül az adott részegységből, és ezen osztoznak majd az importálási pontok.
- **NonShared** — Minden egyes importhoz külön, saját példányt hoz létre a MEF.

Vagyis alapértelmezés szerint a MEF újrafelhasználja a korábban már példányosított részeket.

Az alábbi kódrészletben jelezzük a MEF felé, hogy a **MainDataViewModelek** importálásánál mindig saját, külön példányt szeretnénk:

```
namespace MEFDemo
{
    [Export(typeof(UserControl))]
    public partial class MainDataView : UserControl
    {
        [Import(RequiredCreationPolicy=CreationPolicy.NonShared)]
        public MainDataViewModel ViewModel { get; set; }

        public MainDataView()
        {
            InitializeComponent();
        }
    }
}
```

Alapértelmezett értékek kezelése importáláskor

Importáláskor előfordulhat, hogy egy adott importot a MEF nem tud kielégíteni, azaz nem talál megfelelő exportot az elérhető helyeken. Ilyen esetekben a MEF egy **ChangeRejectedException**-t dob, melynek **Errors** tulajdonságát átvizsgálva a következő hibaüzenet fogad minket: „*No valid exports were found that match the constraint*”

Ha a MEF egy importálási kérelmet nem tud kielégíteni, akkor alapértelmezés szerint dob egy ilyen hibaüzenetet. A hiba elkerülhető, ha engedélyezzük importáláskor az alapértelmezett érték, azaz null érték beállítását arra az esetre, ha nem található megfelelő part az adott importhoz. Ezt az **Import** attribútum **AllowDefault** tulajdonsága segítségével lehet szabályozni:

```
namespace MEFDemo
{
    [Export(typeof(UserControl))]
    public partial class MainDataView : UserControl
    {
        [Import(RequiredCreationPolicy=CreationPolicy.NonShared, AllowDefault=true)]
        public MainDataViewModel ViewModel { get; set; }

        public MainDataView()
        {
            InitializeComponent();
        }
    }
}
```

Metódusok importálása

Eddig kizárólag osztályokat importáltunk, de a MEF képességei nem érnek itt véget! A MEF segítségével szinte bármi importálható, így a függvények is. Az analógia teljes mértékben azonos a korábbiakkal. Meghatározzuk az exportálandó függvényt, az importálás helyéhez pedig egy megfelelő tulajdonságot (**Func<>** típus) készítünk. Az alábbi példa ezt mutatja be.

Az exportálandó függvény:

```

namespace MEFDemo.FunctionImports
{
    public class SpecialAlgorithms
    {
        [Export]
        public IEnumerable DoFiltering(IEnumerable items)
        {
            //Szűrjünk vmilyen logika alapján
            return items...
        }
    }
}

```

Az importálás elvégzése:

```

namespace MEFDemo.ViewModels
{
    [Export]
    public class MainDataViewModel
    {
        IDataSource dataSource;

        [Import]
        public Func<IEnumerable, IEnumerable> FilteringFunction { get; set; }

        [ImportingConstructor]
        public MainDataViewModel([Import]IDataSource dataSource)
        {
            this.dataSource = dataSource;
        }

        public void Filter()
        {
            List<string> items = new List<string> {
                "Arvai Zoltan", "Silverlight 4, Chapter 12", "MEF" };
            var result = FilteringFunction.Invoke(items);
        }
    }
}

```

A **FilteringFunction** tulajdonság típusa **Func<IEnumerable, IEnumerable>**. Ez azt jelzi, hogy olyan függvényt lehet a **FilteringFunction** tulajdonságba betölteni, melynek visszatérési értéke **IEnumerable** típusú, és a paraméterlistán egy paramétert fogad, melynek típusa szintén **IEnumerable**.

Bizonyára feltűnt, hogy nem generikus függvényt használtunk a szűrésre. Sajnos a Silverlight 4.0-ban jelen pillanatban elérhető MEF verzió nem támogatja a generikus metódusok és típusok importálását.

Import kielégítése több part segítségével (*ImportMany*)

Az esetek többségében előfordul, hogy a MEF egyetlen import kielégítésére több megfelelő export definíciót is talál. Ilyen esetekben többnyire egy **ChangeRejectedException**-t kapunk.

Figyelem: Ha az **AllowDefault** paramétert **true**-ra állítottuk valamelyik kulcsfontosságú import attribútumon, előfordulhat, hogy nem kapunk kivételt, csupán az adott importhoz tartozó objektum fa értéke lesz **null**!

A hibát jobban megvizsgálva eljutunk a következő hibaüzenethez: „*More than one export was found that matches the constraint*” Azaz több exportot talált, ami illeszkedne az importhoz. A MEF magától nem

dönthet, a döntést nekünk kell meghoznunk. Ehhez azonban meg kell engednünk a MEF számára, hogy az összes illeszkedő részegységet importálja. Ez az **ImportMany** attribútum segítségével történhet:

```
namespace MEFDemo.ViewModels
{
    [Export]
    public class MainDataViewModel
    {
        IEnumerable<IDataSource> dataSource;

        [Import]
        public Func<IEnumerable, IEnumerable> FilteringFunction { get; set; }

        [ImportingConstructor]
        public MainDataViewModel([ImportMany]IEnumerable<IDataSource> dataSource)
        {
            this.dataSource = dataSource;
        }

        public void Filter()
        {
            List<string> items = new List<string> { "Arvai Zoltan", "Silverlight 4, Chapter 12", "MEF" };
            var result = FilteringFunction.Invoke(items);
        }
    }
}
```

Az importálást követően akár egy **foreach** szerkezettel végig iterálhatunk az adatforrásokon és kiválaszthatjuk a megfelelőt. De vajon honnan tudjuk, hogy melyik a megfelelő? Erre a problémára szolgáltatnak megoldást a metaadatok.

Metaadatok használata

Számos esetben lehet szükség arra, hogy a részegységek különféle információkat továbbítsanak az importálás oldalára. Ezt metaadatok segítségével érhetjük el. Metaadatok készítésére az **ExportMetadata** attribútum segítségével van lehetőség. Közvetlenül string alapú metaadatokat készíthetünk, ezek azonban nem túl kényelmesen használhatók, és fordításidejű típusellenőrzéshez sem biztosítanak támogatást. Szébb megoldás típusos metaadatok készítése! Ehhez első lépésként a megfelelő attribútumra van szükség:

```
namespace MEFDemo.DataSources
{
    [MetadataAttribute]
    [AttributeUsage(AttributeTargets.Class)]
    public class DataSourceAttribute : ExportAttribute
    {
        public DataSourceTypes DataSourceType { get; private set; }

        public DataSourceAttribute(DataSourceTypes dataSourceType)
            : base(typeof(IDataSource))
        {
            DataSourceType = dataSourceType;
        }
    }

    public enum DataSourceTypes { Mock, Live }
}
```

A **MetadataAttribute** jelzi a MEF számára, hogy ez egy metaadatokat tartalmazó attribútum lesz, továbbá az **ExportAttribute**-ből történő származásnak köszönhetően az exportáláskor játszik majd szerepet. Ezzel a metaadattal azt kívánjuk jelezni, hogy az adatforrás éles vagy „mock” adatforrás lesz. Következő lépésként a meta információt el kell helyezni az exporton:

```
namespace MEFDemo
{
    [Export(typeof(IDataSource))]
    [DataSource(DataSourceTypes.Mock)]
    public class SimpleDataSource : IDataSource
    {
        public SimpleDataSource()
        {
            //Inicializálás
            //...
        }
        public IEnumerable GetData()
        {
            //Adatok betöltése
            return null;
        }
    }
}
```

Ugyanez az attribútum szerepelhet a **ComplexDataSource** osztály fölött is, **DataSourceTypes.Live** értékkel.

Importáláshoz és a metaadatok kiolvasásához, egy speciális, az attribútum tulajdonságainak megfelelő interfész definícióra van szükség:

```
namespace MEFDemo.DataSources
{
    public interface IDataSourceMetadata
    {
        public DataSourceTypes DataSourceType { get; set; }
    }
}
```

Nem maradt más hátra, mint az importálás és a metainformációk kiolvasása:

```
namespace MEFDemo.ViewModels
{
    [Export]
    public class MainDataViewModel
    {
        [ImportMany]
        public IEnumerable<Lazy<IDataSource, IDataSourceMetadata>>
            dataSources { get; set; }

        private IEnumerable<IDataSource> GetLiveDataSources()
        {
            var liveDataSources = from p in dataSources
                                  where p.Metadata.DataSourceType == DataSourceTypes.Live
                                  select p.Value;

            return liveDataSources;
        }
        ...
    }
}
```

Az **ImportMany** attribútumhoz tartozó **dataSource** tulajdonság típusa megváltozott. Egy speciális generikus típusból álló generikus listává alakítottuk. A **Lazy<T>** típus azt jelenti, hogy a változó akkor lesz példányosítva, amikor ténylegesen kiértékelésre kerül. Azaz egyfajta „lusta” kiértékelésről beszélünk. A **Lazy<T>** generikus típus első típusparamétere jelen pillanatban az adatforrások típusát jelzi, míg a második típusparamétere a metaadatok formátumát leíró interfészt nevesíti.

A **GetLiveDataSource()** metódus egy LINQ kifejezés segítségével azokat az adatforrásokat választja ki, melyek metaadatuk alapján **DataSourceType.Live** értékkel rendelkeznek.

A **Lazy<T>** típus fontos tulajdonságai:

- **IsValueCreated** – A kifejezés által meghivatkozott objektum példány létrejött-e már?
- **Value** – Az objektum tényleges értéke
- **Metadata** – Kapcsolódó metaadatok

Attribútumok vagy interfészek?

Ez a megoldás roppant elegáns kapcsolódó metaadatok kezeléséhez, de az esetek túlnyomó többségében teljes mértékben elegendő, ha az export típusaként szereplő interfész tartalmazza közvetlenül ezeket az információkat. Ezért extra munkára nincs szükség. Ugyanakkor a **Lazy<T>** típusnak és az attribútum alapú megoldásnak vitathatatlan előnye, hogy az adott típusokból nem készül példány, csak akkor, amikor ténylegesen szükség is van rájuk. Így a fenti példában a „mock” adatforrások nem kerülnek példányosításra, míg a „live” adatforrások igen.

Katalógusok használata

A korábbiakban kizárólag azzal foglalkoztunk, hogy az importálás és az exportálás folyamatát finomítsuk, és magától értetődőnek tekintettük, hogy a MEF megtalálja ezeket a típusokat és az exportokat. Eddig minden munkáért a **CompositionInitializer.SatisfyImports()** metódusa volt a felelős. A háttérben ez a metódus valójában sok munkát végzett. Ezek közül az egyik legfontosabb a katalógusok elkészítése volt.

A Managed Extensibility Frameworkben jeleznünk kell, hogy hol lehet egyáltalán importálásra alkalmas objektumokat felderíteni. Erre a feladatra készültek a különböző katalógusok. A Silverlight az alábbi katalógusokat támogatja:

- **TypeCatalog**
- **AssemblyCatalog**
- **AggregateCatalog**
- **DeploymentCatalog**

A TypeCatalog használata

A **TypeCatalog** objektum egy típusokból álló listát reprezentál. A MEF **TypeCatalog** használata esetén ebből a típuslistából keres megfelelő exportokat:

```
TypeCatalog typeCatalog = new TypeCatalog(typeof(MainDataView),  
    typeof(SimpleDataSource), typeof(ComplexDataSource), ..., ...);
```

Az AssemblyCatalog használata

AssemblyCatalog segítségével egy assemblyt leíró objektumot kell definiálni. A MEF **AssemblyCatalog** használata esetén a hivatkozott assemblyben található típusok között keres megfelelő exportokat:

```
AssemblyCatalog assemblyCatalog = new AssemblyCatalog(Assembly.GetExecutingAssembly());
```


Az *AggregateCatalog* használata

Az **AggregateCatalog** egy katalógusgyűjtemény. Önálló betöltési logikával nem rendelkezik, csupán több különböző katalógust képes összefogni, ezáltal egy időben, egyszerre több forrást képes reprezentálni:

```
TypeCatalog typeCatalog = new TypeCatalog(typeof(MainDataView),
    typeof(SimpleDataSource), typeof(ComplexDataSource));

AssemblyCatalog assemblyCatalog = new AssemblyCatalog(Assembly.LoadFrom(...));

AggregateCatalog aggregateCatalog = new AggregateCatalog();
aggregateCatalog.Catalogs.Add(typeCatalog);
aggregateCatalog.Catalogs.Add(assemblyCatalog);
```

A *DeploymentCatalog* használata

A **DeploymentCatalog** kétségkívül az egyik leghasznosabb katalógus Silverlightban. Segítségével XAP fájlokat tölthetünk le, és a bennük elhelyezett assemblyket tudjuk dinamikusan becsatolni alkalmazásunkba:

```
DeploymentCatalog catalog =
    new DeploymentCatalog(new Uri("http://myslexensions.com/extensions.xap"));
catalog.DownloadProgressChanged +=
    new EventHandler<DownloadProgressChangedEventArgs>(catalog_DownloadProgressChanged);
catalog.DownloadCompleted +=
    new EventHandler<AsyncCompletedEventArgs>(catalog_DownloadCompleted);

catalog.DownloadAsync();
```

A **DeploymentCatalog** aszinkron működésű, így feliratkozhatunk a folyamatos tájékozáshoz a **DownloadProgressChanged**, illetve a **DownloadCompleted** eseményekre is. Az XAP fájl letöltését a **DownloadAsync()** metódus hívásával kezdetjük el.

DeploymentCatalog használatánál könnyen futhatunk **ChangedRejectedException**-be. A kompozíció lehet, hogy már rég előállt, mire az XAP fájl letöltődik. Ilyenkor ezt a katalógust is be kell csatolni, és újra kell építeni a kompozíciót. Sajnálatos módon alapértelmezés szerint ezt az importok nem támogatják, külön kell jelezni az **Import** attribútum **AllowRecomposition** tulajdonság segítségével:

```
namespace MEFDemo
{
    [Export(typeof(UserControl))]
    public partial class MainDataView : UserControl
    {
        [Import(AllowRecomposition=true)]
        public IDataSource DataSource { get; set; }

        public MainDataView()
        {
            InitializeComponent();
        }
    }
}
```

Katalógusok betöltése

A tényleges munkát a **CompositionContainer** osztály végzi. Ő felelős a katalógusok kezeléséért és a részegységek betöltéséért is. A konténer létrehozásakor átadható a katalógus, amelyből a konténernek dolgoznia kell. Ezt a szerepet többnyire egy **AggregateCatalog** tölti be. A **ComposeParts()** hívás végzi el az importálást. A **CompositionInitializer.SatisfyImports()** metódusa lényegében a fenti

katalógusokat és ezt a konténert használja egy alapértelmezett konfigurációban, ahol az összes meghivatkozott assemblyben végez keresést.

```
CompositionContainer container = new CompositionContainer(catalog);  
container.ComposeParts(this);
```

Megjegyzés: A fentiek csupán a MEF deklaratív oldalát mutatták be, ahol az importálás és az exportálás attribútumok alapján történik. Valójában minden exportált objektum felderítésre kerül, az **Import** attribútum létezésétől függetlenül. Az ilyen be nem csatolt partok elérésére a **CompositionContainer** példány **GetExportedValue()**, **GetExportedValues()** függvényei segítségével nyílik lehetőség.

Managed Extensibility Framework a valódi világban

A Managed Extensibility Framework egyszerű és hatékony működésének köszönhetően rendkívüli népszerűségnek örvend a fejlesztői társadalomban. Sokan minden modularizált, generikus megoldásukat MEF segítségével implementálják. Jó példa erre a design-time és az éles ViewModelek közötti dinamikus csere MEF segítségével. Amellett, hogy kétségtelenül töretlen a népszerűsége, a MEF sem egy minden problémára megoldást nyújtó eszköz. Ne feledjük, a MEF elsődleges célja a kiterjeszthető megoldások, a plug-in jellegű szituációk támogatása és nem az alkalmazás struktúrájának főlegesen bonyolítása. Mint minden dinamikus megoldás, a MEF is kisebb negatív hatással lehet az alkalmazás teljesítményére — az importálások végrehajtásának ideje alatt. Használata azonban nagymértékű dinamizmust és lazán csatoltságot hozhat a megoldásainkba, melynek jótékony hatása, előnye vitathatatlan. Mindenképp érdemes elgondolkodni, hogy mikor és hogyan érdemes alkalmazni a MEF-et saját megoldásainkban.

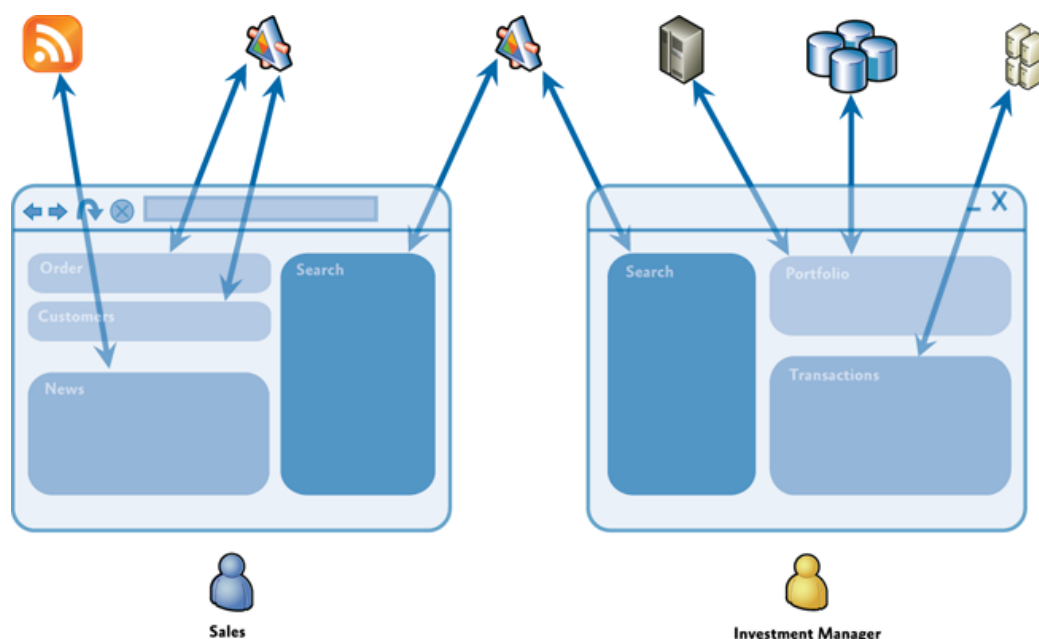
A Composite Application Guidance (PRISM)

A fejezet korábbi részében a fókusz modulok dinamikus betöltésére helyeztük. Ebben a Managed Extensibility Framework komoly segítséget, infrastruktúrát biztosított. A modulok és kiterjesztések memóriába töltése azonban csak az első lépés! Ezeket a modulokat rendkívül gondosan kell megtervezni, hogy akár egymással együtt, de külön-külön is tökéletesen működjenek. A modulokat a MEF tárgyalásakor teljes mértékben általános modulként kezeltük. A következőkben a megjelenítési rétegre fókuszálunk, és azon belül is a fő célkitűzésünk az ún. *kompozit alkalmazások* építése lesz.

Kompozit (összetett) alkalmazásoknál a felhasználói felületek és a kapcsolódó funkcionalitások modulokba szervezése természetes igény. A modularizáció — jól tudjuk — számos előnnyel kecsegtet, nincs ez másképp a felhasználói felület vonatkozásában sem. Korábban az MVVM architektúráis mintával foglalkozó fejezetben már megismertük, hogy a felhasználói felület felosztása kisebb egységekre, a nézetekbe tördelés komoly segítség már kisebb alkalmazások esetén is. Nagyobb alkalmazások megvalósításánál azonban ez a fajta felosztás bár elengedhetetlen, de jó eséllyel túlságosan finom, rengeteg apró komponenst eredményez. Éppen ezért ezek a közös funkcionalitással rendelkező, egy logikai vagy fizikai egységbe tartozó nézetek könnyen és kényelmesen szervezhetők nagyobb egységekbe, ún. *kompozit nézetekbe*. Itt ezekkel a modulokkal foglalkozunk.

A *Composite Application Guidance* (Prism) a Microsoft Patterns and Practices csapata által készített osztálykönyvtár, amely a kompozit alkalmazások fejlesztéséhez tartozó infrastruktúrát biztosítja. A csomagot eredetileg a Windows Presentation Foundationhoz készítették, sőt a csomagnak régebbi múltja is van, a Composite Application Block, amely még a Windows Forms technológiához készült. A könyv írásának pillanatában a Prism 4.0-ás verziója a legfrissebb, amely elérhető a Silverlight 4.0, a Windows Phone 7 és a WPF 4.0 számára is.

A kompozit alkalmazások sematikus szerkezetét a 12-1 ábra mutatja be.



12-5 ábra: Kompozit alkalmazás, több különböző back-end rendszerrel

A PRISM által megcélzott problémák

A PRISM kialakítása során nagyon fontos szerepet játszott, hogy olyan keretrendszert építsenek, amely az ilyen moduláris, kompozit alkalmazások fejlesztéséhez szükséges infrastruktúrát biztosítani tudja. Így tervezői kiemelt figyelmet fordítottak az alábbiakra:

- a modulok kezelése
- a nézetek kezelése
- a shell (keret) kialakítása
- az MVVM támogatása Commanding eszközökkel
- a modulok közötti kommunikáció
- a szolgáltatások hozzáférhetősége

Modulok tervezése és implementálása

A modulok kezelése kiemelt fontosságú téma a Prismben. A modulokat úgy kell megtervezni, hogy biztosítsuk a dinamikus betöltést, az inicializálást, a modulok közös kezelését (absztrakció). Ehhez egy DI keretrendszerre van szükség. A Prism korábbi verzióiban ezt a szerepet kizárólag a Microsoft Unity töltötte be, bár a Prism architektúrális felépítésének köszönhetően lehetőséget biztosított egyéb DI konténerek alkalmazására is. A Prism 4.0-ás verziójában a modulok dinamikus betöltését a Unity 2.0-ás verzióján kívül elvégezhetjük a fejezet korábbi részében megismert Managed Extensibility Framework segítségével is.

A Shell és a Bootstrapper elkészítése

A Prism osztálykönyvtárban az inicializálásért, a *shell* megjelenítéséért és betöltéséért, a modulok létrehozásáért és betöltéséért az ún. **Bootstrapper** objektum a felelős. Attól függően, hogy Unityt, vagy MEF-et szeretnénk használni, választanunk kell a **UnityBootstrapper** és a **MefBootstrapper** között. Ezek az őssztályok a **Microsoft.Practices.Prism.dll**-ben találhatók.

Saját **Bootstrapper** készítése **MefBootstrapper** alapján:

```
namespace PrismDemo
{
    public class Bootstrapper : MefBootstrapper
    {
        // Shell dinamikus betöltése MEF-fel
        protected override DependencyObject CreateShell()
        {
            return this.Container.GetExportedValue<Shell>();
        }

        // Shell objektum beállítása RootVisual-ként
        protected override void InitializeShell()
        {
            base.InitializeShell();

            App.Current.RootVisual = (UIElement)this.Shell;
        }

        // Katalógusok konfigurálása
        protected override void ConfigureAggregateCatalog()
        {
            base.ConfigureAggregateCatalog();

            this.AggregateCatalog.Catalogs.Add(
                new AssemblyCatalog(Assembly.GetExecutingAssembly()));
        }
    }
}
```

Az első fontos lépés a MEF katalógus beállítása. Ezt legegyszerűbben a **MefBootstrapper** osztály **ConfigureAggregateCatalog** metódusának felülírásával érhetjük el. A fenti kódrészletben az aktuális assembly alapján készült **AssemblyCatalog**-ot helyezük el a katalógusgyűjteményben, így először ebben az assemblyben keressük a modulokat.

A második fontos lépés a Shell objektum létrehozása. Ez lesz az a keretvezérlő, amelybe majd az egyes kompozit nézeteket elhelyezzük. Így a Shell objektum gyakorlatilag egy **UserControl**. (A korábbi **MainPage** helyett.) Ezt a **MefBootstrapper** objektum **CreateShell()** metódusának felüldefiníálásával érhetjük el. A Shell példányt a **CompositionContainer** segítségével állítjuk elő.

Harmadik lépésként be kell állítani a Shell-t **RootVisual**-ként, erre a legalkalmasabb hely az **InitializeShell()** metódus.

A fentieknek megfelelően az **App** osztály **Application_Startup** eseményét módosítanunk kell az alábbi módon:

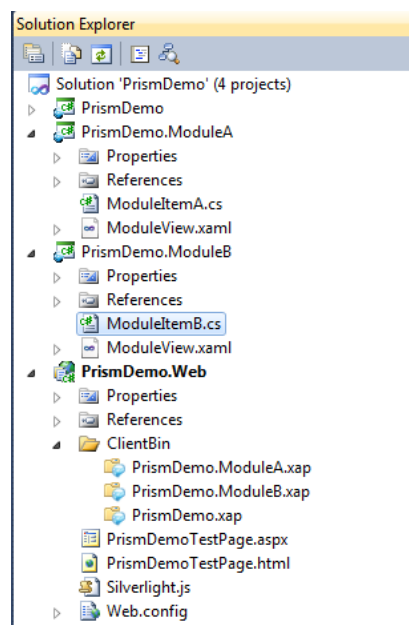
```
Application_Startup(object sender, StartupEventArgs e)
{
    //MainPage helyett Shell-t használunk
    //this.RootVisual = new MainPage();

    //A bootstrapper-t futtatjuk
    Bootstrapper bootstrapper = new Bootstrapper();
    bootstrapper.Run();
}
```

Modulok készítése

A modulokat, ha igazán lazán csatolt megoldást szeretnénk, célszerű külön XAP fájlokban elhelyezni. Ennek további pozitív hatása a használat alapú betöltés. Azaz, egy modult csak akkor töltünk le, ha azt ténylegesen használni is szeretnénk.

Ahhoz, hogy a modulok XAP állományba kerüljenek, Silverlight Application típusú sablon segítségével kell létrehozni őket. A sablonban alapértelmezetten megjelenő **App.xaml** és **MainPage.xaml** fájlok nem fognak a továbbiakban kelleni, így törölhetők. A 12-2 ábrán látható a projekt felépítése.



12-2. ábra: Kompozit alkalmazás projektstruktúrája

Figyeljük meg a 12-2 ábrán, hogy a **ClientBin** mappában megjelennek az XAP fájlok! A **ModuleA** és **ModuleB** projektek nem hivatkoznak egymásra, és a **PrismDemo** alkalmazás projekt sem hivatkozza meg őket.

A modulokat objektumként a **ModuleItemA**, illetve a **ModuleItemB** osztályok fogják reprezentálni. A modulok rendelkeznek a **ModuleExport** attribútummal, amely jelzés a Prism számára, hogy ez az osztály egy modul betöltéséért felelős. Ezenfelül a modulok implementálják az **IModule** interfészt, amelyen keresztül tud a Prism majd kommunikálni a modulokkal. A MEF használatához Prism alatt szükség van a **Microsoft.Practices.Prism.MefExtensions.dll**-re és a **System.ComponentModel.Composition.dll**-re is.

A **ModuleB** lehetséges implementációja:

```
namespace PrismDemo.ModuleB
{
    [ModuleExport(typeof(ModuleItemB), InitializationMode=InitializationMode.WhenAvailable)]
    public class ModuleItemB : IModule
    {
        // Modul objektum létrehozása ezzel a konstruktorral fog történni
        [ImportingConstructor]
        public ModuleItemB()
        {
        }

        // A modul inicializása itt történhet
        public void Initialize()
        {
        }
    }
}
```

A **ModuleExport** második paramétere az **InitializationMode.WhenAvailable** értékkel meghatározza, hogy a modul betöltése azonnal megtörténhessen. Másik lehetőség a használat alapú, **OnDemand** mód lehetne.

Modulok felderítése

Fontos kérdés a továbbiakra nézve, hogy miként találhat rá a **Bootstrapper** az egyes modulokra. A korábbi szekciókban megismert katalógusok, mint például a **DeploymentCatalog** használata teljes mértékben megállja a helyét. Ennél azonban könnyebb és elegánsabb megoldás lehet egy **ModuleCatalog** objektum készítése, amely a háttérben szintén **DeploymentCatalog**-ot használ.

A **ModuleCatalog.xaml** fájl tartalma:

```
<Modularity:ModuleCatalog
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  xmlns:Modularity="clr-namespace:Microsoft.Practices.Prism.Modularity;
    assembly=Microsoft.Practices.Prism">
  <Modularity:ModuleInfoGroup InitializationMode="WhenAvailable">
    <Modularity:ModuleInfo Ref="ModuleA.xap" ModuleName="ModuleItemA"
      ModuleType="PrismDemo.ModuleA.ModuleItemA,
        PrismDemo.ModuleA, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null" />
    <Modularity:ModuleInfo Ref="ModuleB.xap" ModuleName="ModuleItemB"
      ModuleType="PrismDemo.ModuleB.ModuleItemA, PrismDemo.ModuleB, Version=1.0.0.0,
      Culture=neutral, PublicKeyToken=null"/>
  </Modularity:ModuleInfoGroup>
</Modularity:ModuleCatalog>
```

A **ModuleCatalog** konfigurációs állományt egy **ModuleCatalog.xaml** fájl képviseli. A fenti kódrészletben az egyes modulokat **ModuleInfo** objektum segítségével reprezentáljuk. A fenti XAML-ből elő kell állítani egy **IModuleCatalog** példányt. A XAML betöltése a **ModuleCatalog.CreateFromXaml()** hívás segítségével történhet. A modulok betöltéséért a **Bootstrapper** a felelős, így a **Bootstrapper** osztályban felüldefiniáljuk a **CreateModuleCatalog()** metódust:

```
protected override IModuleCatalog CreateModuleCatalog()
{
    return Microsoft.Practices.Prism.Modularity.ModuleCatalog.CreateFromXaml(
        new Uri("/PrismDemo;component/ModuleCatalog.xaml", UriKind.Relative));
}
```

A modulok a fenti módosítások után betöltődnek.

Megjegyzés: A modulok esetén a Prism dll-ekre történő hivatkozásnál a Copy Local=false értéket állítsuk be, különben importálási hibába futhatunk!

Kompozit nézetek megjelenítése

A modulok betöltésre kerülnek, de semmi nem történik még jelen pillanatban. A moduljaink feladata az lesz, hogy elkészítsék a megfelelő nézeteket, és jelezzék a Shell felé. A modulokat most egy egyszerű nézet fogja reprezentálni.

A **ModuleA** által használt **ModuleView** nézet:

```
<UserControl x:Class="PrismDemo.ModuleA.ModuleView"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006">
```

```

mc:Ignorable="d"
d:DesignHeight="300" d:DesignWidth="400">

<Grid x:Name="LayoutRoot" Background="White">
    <TextBlock Text="Hello Module A" FontSize="20"/>
</Grid>
</UserControl>

```

A modul az inicializálása pillanatában egy ilyen nézetet készít és jelzi a Shell felé, hogy az elhelyezhesse. Hogyan tud a modul jelezni a Shellnek és hova rakja a Shell ezt a nézetet?

A megoldást a **RegionManager** objektum nyújtja. A Shellen bizonyos vezérlőket meg lehet jelölni régióként. A **RegionManager** pedig névegyezés esetén a megfelelő vezérlőben helyezi majd el a tartalmat. Ilyen objektum lehet például az **ItemsControl**, a **TabControl**, a **ListBox**, és így tovább.

```

<UserControl x:Class="PrismDemo.Shell"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:regions="clr-namespace:Microsoft.Practices.Prism.Regions;⚡
        assembly=Microsoft.Practices.Prism"
    mc:Ignorable="d"
    d:DesignHeight="480" d:DesignWidth="640">

    <Grid x:Name="LayoutRoot" Background="White">
        <ItemsControl regions:RegionManager.RegionName="MainRegion"/>
    </Grid>
</UserControl>

```

A fenti kódrészletben az **ItemsControl** vezérlőt a **RegionManager** segítségével elneveztük „MainRegion”-nek. Bármelyik vezérlő, aki a MainRegionbe szeretne kerülni, ebben az **ItemsControl**-ban jelenik majd meg.

Nincs más hátra, mint a modulok felkészítése. A moduloknak hivatkozniuk kell a **RegionManager**-re, így a konstruktorokat ennek megfelelően módosítjuk. Az **IRegionManager** példány beinjektálását a MEF végzi majd el. A **RegionManager** példány segítségével az adott nézetet a **Regions** gyűjteményhez hozzá kell adnunk. A felvételnek a régiónév megadásával kell történnie:

```

namespace PrismDemo.ModuleA
{
    [ModuleExport(typeof(ModuleItemA), InitializationMode=InitializationMode.WhenAvailable)]
    public class ModuleItemA : IModule
    {
        IRegionManager regionManager;

        // Az IRegionManager példány átadásáért a MEF lesz a felelős
        [ImportingConstructor]
        public ModuleItemA(IRegionManager regionManager)
        {
            this.regionManager = regionManager;
        }
    }
}

```

```
// A modul inicializása itt történhet
public void Initialize()
{
    // Nézet létrehozása
    ModuleView view = new ModuleView();
    // Nézet bejegyztrálása az adott régióba
    regionManager.Regions["MainRegion"].Add(view);
}
}
```

A Shell kialakítását, elrendezését a **RegionManager** objektum segítségével elvégezhetjük, így a nézetek mindig a megfelelő helyre kerülnek majd betöltésre.

Kommunikáció modulok között

A modulok önálló egységek, melyek nem hivatkoznak egymásra. Információt mégis cserélniük kell! A Prism erre a problémára vezette be az **EventAggregator** elnevezésű objektumokat. Segítségükkel modulok közötti speciális eseményeket készíthetünk, melyek bekövetkezését jelezhetjük, és amelyekre bárhol feliratkozhatunk. Ehhez csupán egy közös csatornára, az **EventAggregator**-ra van szükség. A csatornára az **IEventAggregator.GetEvent<EventType>** metódus által visszaadott objektum **Publish(object)** metódusával publikálhatunk, míg a feliratkozás a **Subscribe(Action<object>)** metódus hívásával lehetséges. Az **EventAggregator** csatorna megszerzése a **RegionManager**-hez hasonlóan, importálással történhet.

A **ModuleA**-ban a **BookRemoved** eseményre az alábbi kóddal iratkozhatunk fel:

```
namespace PrismDemo.ModuleA
{
    [Export]
    public partial class ModuleView : UserControl
    {
        IEventAggregator eventAggregator;

        [ImportingConstructor]
        public ModuleView(IEventAggregator eventAggregator)
        {
            InitializeComponent();
            this.eventAggregator = eventAggregator;
            this.Loaded += new RoutedEventHandler(ModuleView_Loaded);
        }
        void ModuleView_Loaded(object sender, RoutedEventArgs e)
        {
            eventAggregator.GetEvent<BookRemovedEvent>().Subscribe(book =>
            {
                //book.BookID alapján a könyv törlése a listából...
            });
        }
    }
}
```

A **ModuleB**-ben váltjuk ki a **BookRemoved** eseményt:


```

namespace PrismDemo.ModuleB
{
    [Export]
    public partial class ModuleView : UserControl
    {
        IEventAggregator eventAggregator;

        [ImportingConstructor]
        public ModuleView(IEventAggregator eventAggregator)
        {
            InitializeComponent();
            this.eventAggregator = eventAggregator;
        }

        void Button_Click(object sender, RoutedEventArgs e)
        {
            eventAggregator.GetEvent<BookRemovedEvent>().Publish(
                new Book { BookID = 1 });
        }
    }
}

```

A **BookRemovedEvent** osztály kódja:

```

public class BookRemovedEvent : CompositePresentationEvent<Book>
{
    public int BookID { get; set; }
}

```

A csatornára helyezett eseménynek a **CompositePresentationEvent<T>** osztályból kell származnia. Az **EventAggregator**-ok koncepciójukat illetően nagymértékben hasonlítanak az MVVM Light Toolkit Messaging infrastruktúrájára.

MVVM támogatás a Prismben

A Prism is az MVVM alapú alkalmazásfejlesztést szorgalmazza. Az MVVM minta implementálásának egyik kulcsfontosságú gyakorlati kérdése a *commanding* támogatás. A Prismben ezt a **DelegateCommand** objektum segítségével alakíthatjuk ki. A **DelegateCommand** működése és használata megegyezik az MVVM Light Toolkitben megismert **RelayCommand** objektum használatával.

Összefoglalás

A moduláris alkalmazásfejlesztés figyelmet és némi extra munkát igényel, legalábbis az alkalmazás keretének, infrastruktúrájának kialakítását illetően. Ugyanakkor vitathatatlan előnyökkel rendelkezik, akár a tesztelésről, akár a szeparációról vagy csapatmunkáról beszélünk. A moduláris, lazán csatolt rendszerek kialakításában hasznát vesszük külső dependency injection (DI) keretrendszereknek, mint például a Microsoft Unity.

A Managed Extensibility Framework is képes ezt a feladatot ellátni, de fő célja mindenképpen a kiterjeszthetőség támogatása. Ezek az eszközök a modulok dinamikus betöltésére fókuszálnak. A Prism ugyanakkor az előbb említett technológiákat felhasználva egy komplex keretrendszert biztosít a kompozit, moduláris alkalmazások fejlesztéséhez. A modulok betöltésén túl segítséget nyújt a modulok közötti kommunikáció megteremtésében, a kompozit nézetek elrendezésében és az MVVM architektúra implementálásában is. Nagyobb méretű alkalmazások fejlesztése esetén érdemes mérlegelni a könyvtár használatát.

13. A HTML5, az alternatív webes technológia a Silverlight mellett

A korábbi fejezetekben megismerkedhettünk a manapság egyre népszerűbb kliens oldali webes technológiával, a Silverlighttal. Ez a fejezet merőben másról fog szólni: egy olyan webes technológia bemutatásáról, ami a Silverlighttal együtt alkalmazva nagymértékben megkönnyítheti a webes alkalmazások fejlesztését. Ám mielőtt mélyebben beleásnánk magunkat a HTML5 technológiák nyújtotta funkciókba, ismerkedjünk meg azzal, hogy milyen lehetőségek állnak rendelkezésünkre webes alkalmazások fejlesztésére!

Webes környezetben kliens oldali fejlesztésről akkor beszélhetünk, amikor olyan alkalmazásokat kell készíteni, amelyeket a böngészőnk segítségével jelenítünk meg és futtatunk le. Az ilyen alkalmazásokat két nagy csoportra bonthatjuk: vékony- és vastagkliensekre. Vékonyklienseknek nevezzük azokat az alkalmazásokat, amelyek sztenderd webes elemekből épülnek fel: HTML leíró nyelv, CSS stílusleíró nyelv és JavaScript. Vastagklienseknek pedig azokat, amelyeknek az alkalmazás futtatásához a böngészőnek külső bővítményre van szüksége.

A sztenderd HTML megoldás azzal a nagy előnnyel rendelkezik a vastagkliensekkel szemben, hogy bármilyen környezetben és eszközön futtatható, függetlenül a megjelenítő eszköz típusától (lehet hagyományos asztali számítógép, kézi számítógép vagy telefon) vagy az eszköz operációs rendszerétől. Az egyetlen követelmény csupán annyi, hogy az eszköz rendelkezzen böngészővel. A Silverlightről azonban ugyanez nem mondható el, hiszen a böngészőnek rendelkeznie kell Silverlight bővítménnyel. Ezt a bővítményt számtalan eszköz esetében lehetetlen biztosítani (pl. nem Windows alapú okos telefonok). A HTML másik nagy előnye (és talán hátránya is), hogy lényegesen egyszerűbb programozni, mint a Silverlightot, hiszen csupán egy egyszerű leíró nyelvről van szó, kiegészítve JavaScript támogatással.

A fenti sorok olvasása után feltehetjük magunkban azt a kérdést, hogy a HTML előnyeit figyelembe véve miért van szükség pl. a Silverlightra? A válasz nagyon egyszerű! Vannak olyan feladatok és funkciók, amelyeket tisztán HTML környezetben lehetetlen megoldani, akár kivitelezési, akár biztonsági szempontból. Vegyünk példának egy üzleti alkalmazást: tervezéskor mindenképp többretekű alkalmazásban kell gondolkodni, ami elboldogul az összetett objektum rendszerrel, üzleti logikával és folyamatokkal, és az adatokat is biztonságosan kezeli. Mindezt HTML környezetben megvalósítani szinte lehetetlen: az objektumrendszer és az üzleti logika programozása akadályokba ütközhet a JavaScript nyelv hiányosságai miatt. Bizonyos esetekben kritikus lehet, hogy a felhasználónak rálátása van az alkalmazás forráskódjára és az alkalmazás mögött meghúzódó üzleti logikára. Összességében sem a HTML, sem a Silverlight nem jobb a másiknál, hiszen azok más-más célt szolgálnak. A két technológia inkább egymás kiegészítéseként fogható fel, mintsem egymás riválisaként.

A HTML egy közel két évtizeddel ezelőtt született leíró nyelv, és ez idő alatt számtalan változáson ment át — ennek köszönhetően több változat is létezik belőle. A számos verzió közül a legfrissebb a HTML5 nevet kapta. Még mielőtt mélyebben beleásnánk magunkat, fontos megjegyezni, hogy a fejezet írásakor a szabvány még nincs teljesen befejezve, viszont az összes elterjedt böngésző már most támogatja bizonyos részeit, és mindemellett a webes szabványokat koordináló konzorcium szerint a webes fejlesztés következő lépcsőfoka. A korábbi változatokhoz képest lényeges a változás: az új szabvány nemcsak azt írja elő, hogy hogyan kell a tartalmat szabványos XML sémából előállítani, hanem azt is, hogy ha a tartalom nem felel meg az XML sémának, akkor hogyan kell eljárni. Ennek köszönhetően várhatóan a böngészők közötti kompatibilitási gondok is megszűnnek majd. A szabvány célja, hogy a HTML5 technológiák a webes igényeket teljes mértékben kielégítsék (pl. multimédiás tartalmak megjelenítése, ábrák, alakzatok és animációk rajzolása). Az új webes trend négy fő részből tevődik össze:

- a dokumentumot leíró HTML nyelv,
- a dokumentum stílusát szabályozó CSS stílusleíró nyelv,
- a programozási háttérrel biztosító JavaScript nyelv és
- a HTML5-höz kapcsolódó modulok.

A fejezet további részében ezekkel az alappillérekkel fogunk megismerkedni.

HTML5 alapok

Gyakorlati szempontból a HTML5 újdonságai nagyon izgalmasak, hiszen a változásokat már a nyelv elemei között észrevehetjük: a régi vezérlő elemek közül számos nem használt elem kikerült a szabványból, a megmaradt elemek nagy részének jelentését újrafogalmazták, továbbá számtalan olyan új szemantikus elem került a szabványba, amelyek inkább a tartalom jellegére utalnak, mintsem az adott elem szerkezeti felépítésére. Az új szemantikus elemek segítségével meghatározhatjuk, hogy egy-egy elem milyen tartalmi funkciót fog betölteni a dokumentumunkban: navigációs sáv, fejléc, stb. A szemantikus elemeken túl új multimédiás vezérlő elemekkel is találkozhatunk, amelyek segítségével videót és hanganyagot lehet lejátszani külső bővítmények nélkül. Viszont a legizgalmasabb új vezérlő elem a vászon (**canvas**). Mint ahogy a neve is utal rá, leginkább egy olyan rajzvászonhoz lehet hasonlítani, amire mindenféle grafikai elemet, ábrát és videót lehet rajzolni.

Szemantikus vezérlőelemek

Mint ahogy már említettem, az új szemantikus vezérlő elemeknek fő célja az, hogy mind a böngészők, mind a fejlesztők számára átláthatóbbá tegye a dokumentumot tartalmi és szerkezeti szinten egyaránt. Nézzünk egy gyakorlati példát! Adott egy dokumentum, például egy blog. Az oldalon el szeretnénk helyezni egy navigációs sávot, egy tartalmi részt és egy olyan részt, ahova az olvasók megjegyzéseket fűzhetnek. Az oldal szerkezeti felépítéséhez használhatjuk a hagyományos **<div>** elemeket is, ám ebben az esetben mind a böngésző, mind a forráskódot értelmező fejlesztő számára sokkal nehezebb értelmezni a dokumentumot. Ám abban az esetben, ha az új szemantikus elemeket használjuk, első ránézésre egyértelmű az oldal felépítése.

```
<header>
  <hgroup>
    <h1>My blog...</h1>
    <h3>written by x.y</h3>
  </hgroup>
</header>
<nav>
  <ul>
    <li><a href="#">home</a></li>
    <li><a href="#">blog</a></li>
    <li><a href="#">about</a></li>
  </ul>
</nav>
<article>
  <header>
    <!--Hagyományos szöveges mezőként fog megjelenni a böngészőn -->
    <time datetime="2011-01-26">January 26, 2011</time>
    <h1>Post Title</h1>
  </header>
  <p>Post content...</p>
  <figure></figure>
</article>
```

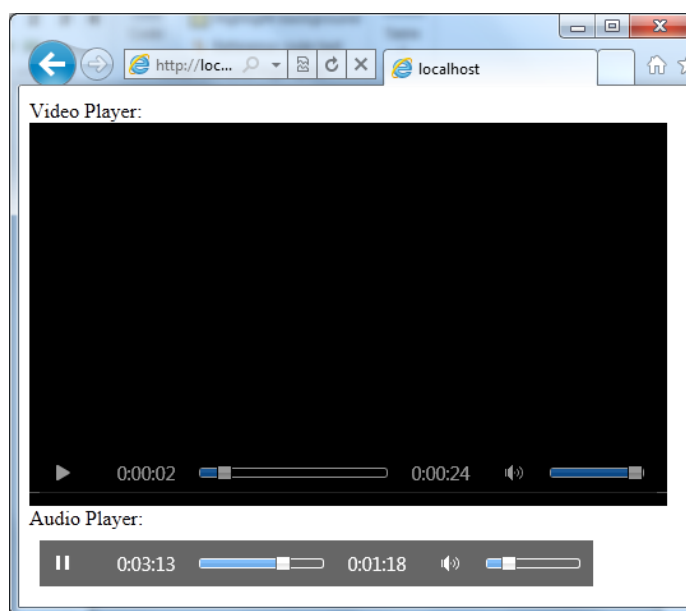
Multimédiás vezérlőelemek

Az internetet böngészve gyakran találkozhatunk olyan weboldalakgal, amelyekbe valamilyen multimédiás tartalom van beágyazva. Ennek oka az, hogy az utóbbi időben elterjedtebbé vált az efféle beágyazott

tartalom (legfőképp a videó), ami nem meglepő. Ki ne osztaná meg kisfilmét, saját készítésű klipjét vagy a kiránduláson készített felvételeket a barátaival, ismerőseivel? Az egyszerűen használható Flash és Silverlight bővítmények elterjedésével és az internet sávszélesség növekedésével minden akadály elhárult a multimédiás tartalom felhasználás elől. Ám ha ilyen tartalmat szeretnénk megjeleníteni, minden esetben szükség van egy vastagkliensre, ami a tartalmat lejátssza, és egy bővítményre, ami a lejátszót futtatja. Szerencsére a HTML5 specifikációban erre is gondoltak, ezért két új vezérlőelemmel bővítették a felhasználható elemek sorát: a **<video>** és az **<audio>** elemekkel. A két vezérlő ugyanazokkal a tulajdonságokkal rendelkezik (a szélesség és a magasságot leszámítva, ugyanis ennek nincs értelme hanganyag lejátszásakor).

```
<video autoplay controls preload="auto" loop height="300" width="500">
  <source src="myvideo.mp4" type="video/mp4" />
  <source src="myvideo.ogv" type="video/ogg" />
</video>
<audio autoplay controls preload="auto" loop >
  <source src="song.ogg" type="audio/ogg" />
  <source src="song.mp3" type="audio/mpeg" />
</audio>
```

A vezérlőelem **autoplay** tulajdonságával lehet beállítani, hogy a média betöltése után automatikusan induljon el a lejátszás, míg a **controls** tulajdonság megjeleníti a lejátszó vezérlőgombjait. A **preload** tulajdonsággal lehet beállítani azt, hogy az oldal betöltése után a letöltendő tartalom mely részei töltődjenek le automatikusan. Ez háromféle értéket vehet fel: **auto** (a teljes tartalom letöltése), **metadata** (csak a média információk letöltése) és **none** (nincs automatikus letöltés, csak a lejátszás elindításakor töltődik le a tartalom). Mint láthattuk, a **<source>** elemmel lehet meghatározni a tartalom forrását, illetve forrásait. Ha rendelkezésre áll, célszerű több forrást meghatározni eltérő tömörítéssel és formátummal, ugyanis a böngészők más és más formátumokat támogatnak.



13-1 ábra: Beépített multimédia lejátszó

A vászon vezérlőelem

A HTML5 szabvány egyik legizgalmasabb új vezérlőeleme a vászon (**<canvas>**), amelynek segítségével dinamikus, programozható módon jeleníthetünk meg kétdimenziós grafikákat, képeket és alakzatokat a böngészőnkben. Ez az új elem hívatott átvenni az eddig oly népszerűvé vált Flash technológia helyét, hiszen az egyszerűbb képek és alakzatok kirajzolásától az összetettebb animációk programozására is felhasználható, valamint lehetőséget ad a videókkal való munkavégzésre is. Megjelenésével a webes

multimédiás alkalmazások egy új dimenziója tárulhat ki előttünk: a népszerű böngészőben futó fizető webes játékok újjászületése mellett készíthetünk saját, nyílt forráskódú videó lejátszó alkalmazásokat, egyszerűbb animációkat, valamint reklámokat is. Üzleti célokra is felhasználhatjuk, hiszen látványos grafikonokat és diagramokat készíthetünk csupán a böngészőnk segítségével. A vászon további előnye még az is, hogy ugyanazt az animációt rövidebb programkóddal is elérhetjük, mint a Flash vagy Silverlight esetében. Azonban a sok előnye mellett hátránya is akad, hiszen ha animációt készítünk vagy videót játszunk vissza, kézzel kell gondoskodni a vászon újrajzolásáról.

Alapműveletek a vásznon

Mint ahogy már említettem, a vászon legnagyobb előnye, hogy a különféle geometriai alakzatok kirajzolása igen egyszerű. Ám még mielőtt tovább lépnénk, nézzük meg, hogyan is érhetjük el a vászon szolgáltatásait a gyakorlatban! Ehhez két dologra lesz szükségünk: a **<canvas>** elem létrehozására a dokumentumban és a rajzoló környezet (**context**) inicializálására JavaScript oldalon.

```
<!--1. lépés-->
<canvas id="myCanvas" width="600" height="400">
</canvas>

<!--2. lépés-->
<script type="text/javascript">
    var mycanvas = document.getElementById("mycanvas");
    var context = mycanvas.getContext("2d");
</script>
```

Ha ezzel megvagyunk, akkor neki is láthatunk a legegyszerűbb alakzat — a téglalap — megrajzolásának. A kirajzolás előtt célszerű törölni a vászon tartalmát.

Készíteni fogunk egy olyan téglalapot, ami minden irányban 10 pixel távolságra van a vászon széleitől, 200 pixel széles és 100 pixel magas. A téglalap körvonalának megrajzolásához a vászon **strokeRect(posX, posY, width, height)** függvényét, a kitöltéséhez pedig a **fillRect(posX, posY, width, height)** függvényt fogjuk használni:

```
var mycanvas = document.getElementById("mycanvas");
var context = mycanvas.getContext("2d");

// A vászon törlése a művelet megkezdése előtt.
// A rajzolási műveletet mindig a vászon környezete, azaz a context fogja végezni.
// A teljes vászon törléséhez kiindulópontnak a (0,0) pontot kell megadni, méretnek
// pedig a vászon szélességét és magasságát
context.clearRect(0, 0, mycanvas.width, mycanvas.height);

// A téglalap megrajzolása
context.strokeRect(10, 10, 200, 100);

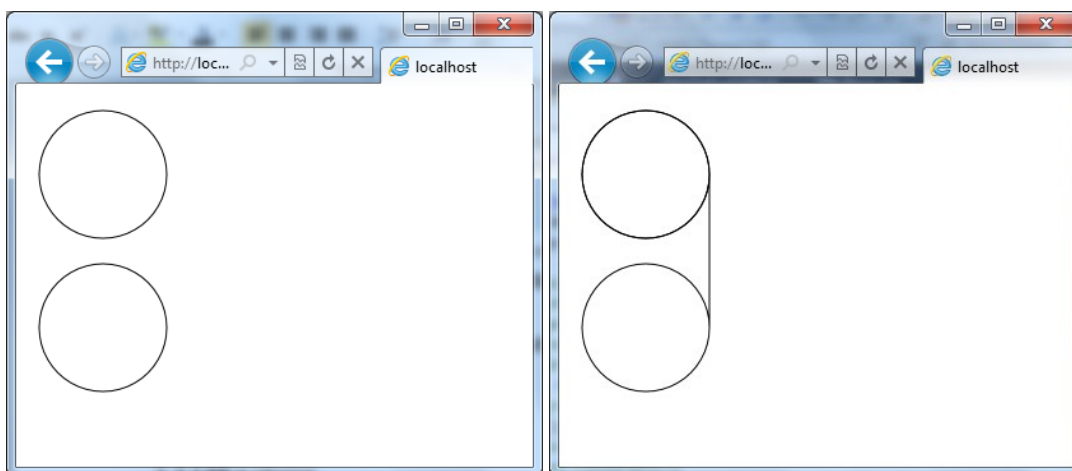
// A téglalap kitöltése
context.fillRect(10, 10, 200, 100);
```

A következő geometriai alakzat, a kör megrajzolása már nem ennyire egyszerű: a vászon **arc(centerX, centerY, startAngle, endAngle)** függvényét kell használni, ami igazából körív megrajzolására való. Viszont a függvény megfelelő paraméterezésével alkalmas kör rajzolására is (a szög kiinduló és végpontját radiánban kell megadni és nem fokban!). A téglalap kirajzolásához képest az a legnagyobb különbség, hogy a függvény csak egy virtuális körívet rajzol, és kézzel kell gondoskodni az alakzat körvonalának megrajzolásáról és kitöltéséről (nemcsak a körív, hanem minden vonal rajzolásakor is hasonló a helyzet). Kitölteni a vászon **fill()**, körvonalat rajzolni pedig a vászon **stroke()** metódusaival lehet. Szintén minden vonalrajzolás esetében elmondható, hogy az alakzat rajzolásának kezdetét és befejezését jelezni kell a vászonnak a **beginPath()** és **closePath()** függvényekkel. Mindezek ismeretében már meg is rajzolhatjuk a körünket:

```
// Alakzat rajzolás elkezdése
context.beginPath();
// 50 pixel sugarú virtuális kör rajzolása, melynek középpontja az (60, 60)-as pont
// A szöveget kiinduló pontját 0 radiánra állítottuk, végpontját pedig 2PI-re, amivel
// egy teljes kört.
context.arc(60, 60, 50, 0, Math.PI * 2, false);
// Körvonal megrajzolása
context.stroke();
// Alakzat rajzolás vége
context.closePath();

context.beginPath();
// Egy másik kör rajzolása a vászon másik pontjába
context.arc(60, 180, 50, 0, Math.PI * 2, false);
context.stroke();
context.closePath();
```

A fenti példában azért rajzoltunk két kört, hogy megértsük, miért is fontos a **beginPath()** és a **closePath()** függvények használata. A 13-2. ábra bal oldala a fenti kód végeredménye, a jobb oldala pedig szintén a fenti kód végeredménye azzal a módosítással, hogy elhagytuk a **beginPath()** és a **closePath()** függvényeket. A jelenség magyarázata roppant egyszerű: a második esetben a böngésző egy alakzatként értelmezi a két kört.

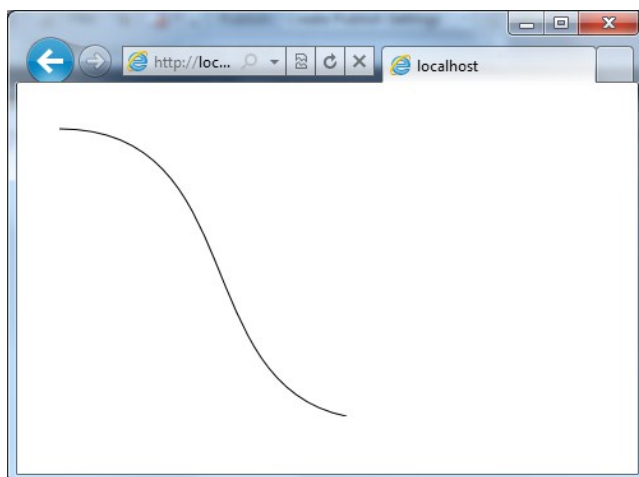


13-2 ábra: Különbség aközött, hogy jelezzük-e a vászonnak az alakzat rajzolásának kezdetét és végét, vagy sem

A körökön és vonalakon túl rajzolhatunk egyszerűbb vonalakat és görbéket egyaránt. Ebben az esetben a vászon **moveTo(x,y)** és **lineTo(x,y)** metódusai lesznek segítségünkre: az előbbivel a virtuális ecsetünket egy adott pontba mozgathatjuk, az utóbbival a virtuális ecset aktuális helyzetétől húzhatunk vonalat a megadott pontba. Egyenes vonalakon túl harmadfokú és negyedfokú Bezier görbéket is rajzolhatunk, mégpedig a vászon **bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)** és **quadraticCurveTo(cpx, cpy, x, y)** függvényeivel. Nézzünk meg egy példát a harmadfokú görbére:

```
context.beginPath();
context.moveTo(25, 25);
context.bezierCurveTo(175, 25, 125, 225, 250, 250);
context.stroke();
context.closePath();
```

A végeredmény a következő:



13-3 ábra: Harmadfokú Bezier görbe

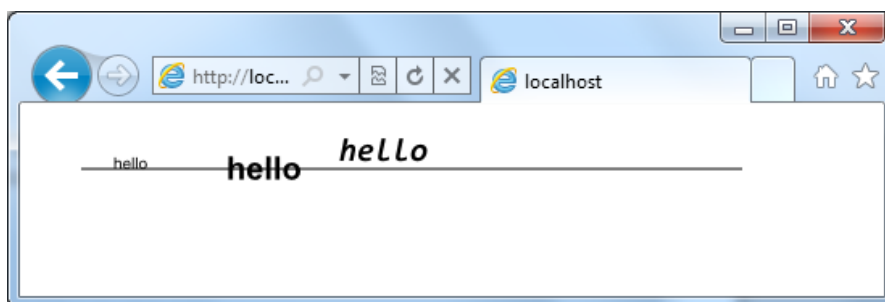
Azt már láthattuk, hogy egyszerű alakzatok vászonra rajzolása pár sorból is megoldható. De mi a helyzet akkor, ha szöveget szeretnénk a vászonra írni? A megoldást a **fillText(text, posX, posY)** és **strokeText(text, posX, posY)** metódusok szolgáltatják. Természetesen módunkban áll meghatározni a kirajzolandó szöveg stílusát és elhelyezkedését. A betűtípust a vászon **font** tulajdonságával lehet módosítani: a CSS-ben használt betűstílus megadási formával. Amennyiben ezt a tulajdonságot nem állítjuk be, a rendszer a „10px sans-serif” betűtípussal fogja megrajzolni a szöveget.

```
// Szöveg egyszerű kirajzolása
context.fillText("hello", 50, 30);

// Szöveg stílusának beállítása (stílus méret betűtípus formátumban),
// elhelyezkedésének beállítása majd a szöveg kirajzolás.
context.textBaseline = "middle";
context.font = "bold 20px Arial ";
context.fillText("hello", 120, 30);

// Új stílus beállítás, majd az új szöveg kirajzolása
context.textBaseline = "bottom";
context.font = "italic bold 20px Consolas ";
context.fillText("hello", 190, 30);
```

A szöveg elhelyezkedésének módját a vászon **textBaseline** tulajdonságával lehet állítani, és a következő értékeket veheti fel: **top**, **hanging**, **middle**, **alphabetic**, **ideographic**, **bottom**. A szemléletesség kedvéért a fenti kódot kiegészítettem egy vízszintes vonallal és két szöveggel, eltérő elhelyezkedéssel. Így a böngészőben megjelenítve a következőt láthatjuk:



13-4 ábra: Szöveg kirajzolása vászonnal

Előfordulhatnak olyan esetek is, amikor mindenképp szükségünk lehet a kirajzolandó szöveg méreteire. Ezt a vászon **measureText(text)** metódusával tehetjük meg.

Körvonal és kitöltés stílusok

Azt már tudjuk, hogy a vásznon az alakzatok kitöltése a **fill()**, **fillText()** és **fillRect()** metódusokkal, a körvonalak megrajzolása pedig a **stroke()**, **strokeText()** és **strokeRect()** függvényekkel történik. De természetesen nemcsak fekete színnel tudunk körvonalat rajzolni és alakzatot kitölteni. Választhatjuk az egyszerű színnel való kitöltést, a színátmenetes kitöltést és a mintával való kitöltést is.

Még mielőtt közelebbről is megismerkednénk ezzel a három lehetőséggel, nézzük meg, hogyan lehet beállítani a kitöltés, illetve a körvonal stílusát! A körvonal színét a vászon **strokeStyle**, a körvonal vastagságát a **lineWidth**, a kitöltés színét pedig a **fillStyle** tulajdonságával lehet megváltoztatni:

```
context.lineWidth = 5;
context.fillStyle = "red";
context.strokeStyle = "green";
context.fillRect(0, 0, 300, 150);
context.strokeRect(0, 0, 300, 150);
```

Abban az esetben, ha a kitöltéshez vagy körvonalhoz egyszerű színt szeretnénk használni, úgy a CSS-ben alkalmazható színmegadási formákat használhatjuk (például **red**, **#FF0000**, **rgb(255,0,0)**, **rgba(255,0,0,0.5)**, **hsl(0,100%,50%)**).

Színátmenetes kitöltés esetén az egyenes- és a sugaras színátmenet közül választhatunk. Mind a két esetben a **fillStyle** illetve **strokeStyle** tulajdonságnak egy **CanvasGradient** objektumot kell átadni. Egyenes színátmenet definiálásához ezt az objektumot a vászon **createLinearGradient(x0, y0, x1, y1)** függvényével hozhatjuk létre, míg sugaras színátmenet definiálásához a vászon **createRadialGradient(x0, y0, r0, x1, y1, r1)** függvényét kell alkalmaznunk. Az első esetben a vászon egy virtuális téglalap, a második esetben pedig két virtuális kör segítségével rajzolja meg a színátmenetet. A színátmenetek csak a megadott virtuális alakzatok koordinátáin belül láthatóak, azon kívül átlátszóak. Tehát ha definiálunk egy kisméretű egyszerű színátmenetet, majd egy nagyobb téglalapot töltünk ki vele, a megrajzott téglalapnak csak egy része lesz kitöltve a színátmenettel. A **CanvasGradient** objektum létrehozását követően már csak hozzá kell rendelni az ún. *színvégpontokat* a színátmenethez, és készen is vagyunk. Színvégpontot a **CanvasGradient** objektum **addColorStop(offset, color)** metódusával adhatunk hozzá a színátmenethez, ami paraméterként a 0 és 1 tartomány közé eső pont helyét és a végpont színét várja.

```
var gradient = context.createLinearGradient(0, 0, 500, 200);
gradient.addColorStop(0.0, "rgba(255,0,0,0.0)");
gradient.addColorStop(0.5, "rgba(255,0,0,0.8)");
gradient.addColorStop(1.0, "rgba(255,0,0,1.0)");

context.fillStyle = gradient;
context.fillRect(0, 0, 500, 200);
```

Mintával való kitöltéskor sincs sokkal nehezebb dolgunk, hiszen csak létre kell hozni egy **CanvasPattern** objektumot a vászon **createPattern(image, repetition)** függvényével, amit később a fentiekhez hasonló módon kell beállítani a vászonnak. Első paraméterként magát a mintát kell meghatározni: átadhatunk képet, egy másik vászon objektumot és videót is. Második paraméterként pedig az ismétlődés módját kell beállítani (a CSS-ben használt **repeat**, **no-repeat**, **repeat-x** és **repeat-y** értékeket).

```
// Mintaként használt kép inicializálása
var patternImage = new Image();
patternImage.src = "background.jpg";

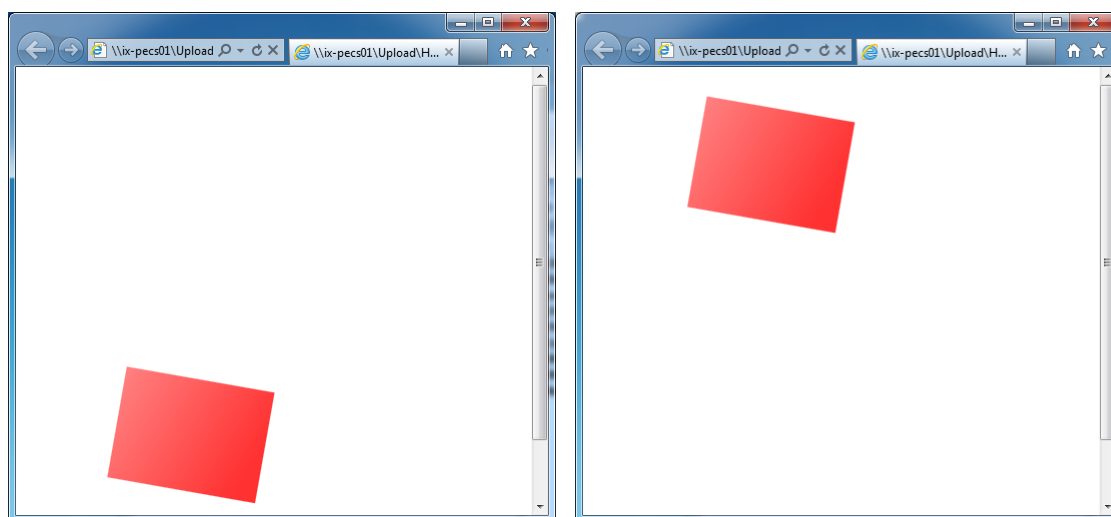
// Csak a kép letöltődése után állítjuk be mintaként a képet,
// ellenkező esetben előfordulhat, hogy a minta üres lesz
patternImage.onload = function () {
```

```
var pattern = context.createPattern(patternImage, "repeat");
context.fillStyle = pattern;
context.fillRect(0, 0, 500, 200);
}
```

Geometriai műveletek a vásznon

Miután már tudunk egyszerűbb alakzatokat és szöveget rajzolni a vásznonra, ideje megismerkedni a vászon egyik legérdekesebb funkciójával, a geometriai transzformációkkal. A vászon legnagyobb ereje talán a geometriai transzformációk animálásában rejlik, hiszen a korábban leírt funkciók és transzformációk kombinálásával látványos vizuális hatásokat érhetünk el csupán pár sornyi kódolás segítségével. A transzformációs függvények ismertetése előtt nézzük meg, hogy a vászon hogyan kezeli, értelmezi és hajtja végre a transzformációkat! A vászon inicializálásakor a háttérben létrejön egy alapértelmezett transzformációs mátrix, majd a különböző transzformációk ezt az alapértelmezett mátrixot fogják megszorozni például az elforgatás mátrixával. Miután minden transzformációt elvégeztünk, a vászon úgy jeleníti meg a képet, hogy alkalmazza rá a módosított transzformációs mátrixot. Természetesen, ha nem állítunk be semmilyen transzformációt, úgy az alapértelmezett mátrixot végrehajtva nem történik semmi. Tekintve, hogy az egyes transzformációs lépések végrehajtása között mátrixszorzás történik, a végrehajtott geometriai műveletek sorrendje nagyon is számít. A végeredmény más lesz, ha a mátrixot eltoljuk, elforgatjuk, majd megint eltoljuk, mintha először eltoltuk volna, majd ismét eltoltuk volna és csak utána forgattuk volna el (13-5 ábra).

Természetesen a vásznon nemcsak egyetlen transzformációt lehet végrehajtani, hanem tetszőleges számút. A transzformáció alkalmazása előtt eltárolhatjuk a rajzvászon állapotát (így a transzformációs mátrixot is) egy speciális veremben a **save()** metódus segítségével, majd a transzformáció végrehajtása után visszaállíthatjuk a kiindulási állapotot a veremből a **restore()** függvénnyel. Arra is lehetőségünk van, hogy egyes transzformációk alkalmazása után visszaállítsuk a vászon transzformációs mátrixát az eredeti állapotába, vagy felülírjuk egy teljesen új mátrixszal. Ebben a **setTransform(a, b, c, d, e, f)** függvény fog segíteni, amely visszaállítja a transzformációs mátrixot az eredeti állapotába, majd alkalmazza rá a paraméterként átadott transzformációt.



**13-5 ábra: Különbségek a transzformáció sorrendjében
(három egyforma transzformáció, más sorrendben)**

A legegyszerűbb transzformációt a vászon **transform(a, b, c, d, e, f)** függvényével érhetjük el. A függvény felülírja a vászon aktuális transzformációs mátrixát úgy, hogy az aktuális mátrixot a megadott mátrixszal megszorozza. A paraméterként megadott mátrix a következőképp fogható fel:

$$\begin{matrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{matrix}$$

Természetesen a mátrix transzformációs függvényeken túl használhatunk olyan egyszerűbb függvényeket is, mint például a **rotate(angle)**, **translate(posX, posY)** és a **scale(width, height)** függvények, amelyekkel elforgathatjuk, eltolhatjuk és átméretezhetjük a vásznat. Ezekben az esetekben is mátrix transzformáció történik, azonban itt nem szükséges a transzformációs mátrixot definiálnunk.

Kép és videó a vásznon

A vászon számos érdekes funkciója közül szintén kiemelendő az, hogy képeket, videókat és más vásznakat tölthetünk be a vászonra, majd a betöltött képekkel különböző műveleteket végezhetünk.

Átméretezhetjük, elforgathatjuk, animálhatjuk őket, valamint a pixelek módosításával látványos vizuális trükköket és hatásokat érhetünk el. A képek betöltése magától értetődő: létrehozunk egy kép objektumot, majd a kép letöltése után azt egyszerűen a vászonra rajzoljuk. De mi a helyzet a videókkal és más vásznnal? Ebben az esetben létre kell hozni a dokumentumban egy **<video>** (vagy **<canvas>**) elemet, és ezt az objektumot kell átadni a vászonnak. Az objektum átadását követően a vászon a videó aktuális képkockájával fog dolgozni. A képek, videók és vásznak betöltésében a **drawImage()** függvény lesz a segítségünkre, amelyet háromféleképpen hívhatunk meg:

- **drawImage(img, posX, posY)**: a paraméterként átadott képet betölti a megadott pozícióba.
- **drawImage(img, posX, posY, width, height)**: a paraméterként átadott képet betölti a megadott pozícióba, valamint átméretezi a megadott szélességnek és magasságnak megfelelően.
- **drawImage(img, clipX, clipY, clipWidth, clipHeight, posX, posY, width, height)**: a betöltendő képet körbevégi a megadott feltételek szerint, majd betölti és átméretezi megadott szélességűre és magasságúra.

És most nézzünk meg egy példát arra, hogyan rajzolunk videót a vászonra:

```
<canvas id="mycanvas" width="640" height="320"></canvas>
<video id="myvideo" width="640" height="320">
  <source src="sample.mp4" type="video/mp4" />
</video>
<script type="text/javascript">
var myvideo = document.getElementById("myvideo");
var mycanvas = document.getElementById("mycanvas");
var context = mycanvas.getContext("2d");

drawCanvas();

// A vászont újrarajzó függvény
function drawCanvas() {
  // Ha vége a videónak, akkor a vásznat se rajzoljuk újra
  if (myvideo.paused || myvideo.ended)
    return;

  context.clearRect(0, 0, 640, 320);
  context.drawImage(myvideo, 0, 0, 640, 320);

  // Vászon frissítése és újrarajzolása 25FPS-el
  window.setTimeout(drawCanvas, 1000 / 25);
}
</script>
```

Amennyiben a vászon pixeleit szeretnénk módosítani, úgy a **createImageData**, **getImageData** és a **putImageData** függvények lesznek a segítségünkre. A **createImageData(width, height)** metódussal egy megadott méretű üres pixelhalmazt hozhatunk létre, a **getImageData(posX, posY, width, height)** metódussal a vászon egy megadott pontjából ragadhatunk ki egy területet, a **putImageData(data, posX, posY)** függvénnyel pedig a vászon adott pozíciójába szűrhetjük be a megadott pixeleket (a meglévőket felülírva). És most egy példán keresztül nézzük meg, hogyan is néz ki ez a gyakorlatban! A vászonra rajzolunk egy piros téglalapot, majd a vásznat szürke árnyalatúvá alakítjuk:

```
// Téglalap megrajzolása
context.lineWidth = 5;
context.strokeStyle = "red";
context.strokeRect(30, 30, 200, 100);
// Eltároljuk a vászon képpontjait egy ImageData objektumban (tömb).
var imageData = context.getImageData(0, 0, 500, 300);
// Ebbe a most még üres ImageData objektumban fogjuk eltárolni a módosított képpontokat
var newImageData = context.createImageData(500, 300);

// Végigszaladunk a képpontokon, négyessével, mert egy képpont négy helynek felel meg a //
// tömbben (három alapszín + átlátszóság)
for (var i = 0; i < imageData.data.length; i += 4) {
    // A képpont piros komponensének kinyerés
    var red = imageData.data[i];
    // A képpont zöld komponensének kinyerés
    var green = imageData.data[i + 1];
    // A képpont kék komponensének kinyerés
    var blue = imageData.data[i + 2];
    // A képpont átlátszóság komponensének kinyerés
    var alpha = imageData.data[i + 3];

    // A képpont RGB komponenseinek súlyozott átlagolása.
    // A hagyományos átlagoláshoz képest ez az eljárás jobban kiemeli az emberi szem
    // által érzékelt színeket, így élesebb lesz a végeredmény
    var avg = red * 0.21 + green * 0.71 + blue * 0.07;

    // Az üres ImageData objektum képpontjait felülírjuk a módosított képpontokkal
    newImageData.data[i] = avg;
    newImageData.data[i + 1] = avg;
    newImageData.data[i + 2] = avg;
    newImageData.data[i + 3] = alpha;
}
// Vászon képpontjainak felülírása a módosított képpontokkal
context.putImageData(newImageData, 0, 0);
```

Azonban fontos megjegyezni, hogy a **getImageData** és **putImageData** függvényeknek nagy a számításigénye, ezért körültekintően kell eljárni (főleg animáció, valamint nagyméretű vászon esetében) alkalmazásuk során. Ettől függetlenül nagyon látványos grafikai hatásokat érhetünk el segítségükkel, csupán pár sornyi kód írásával.

2.4 JavaScript újdonságok

Nem a HTML5 specifikációjához tartoznak a JavaScript programozási nyelvet érintő változások és új funkciók, ám a HTML és a JavaScript szoros kapcsolata miatt mégis érdemes szót ejteni róla ebben a fejezetben. Annak, aki már valaha foglalkozott vékonykliens oldali webes fejlesztéssel, bizonyára nem kell bemutatni, hogy milyen hiányosságai vannak a JavaScript nyelvnek. Ám szerencsére a HTML5 megjelenésével nem csupán a használható HTML elemek köre bővült ki és alakult át, hanem JavaScript oldalon is történtek olyan változások, amelyek megkönnyítik a fejlesztést és a „HTML programozását”.

A legérdekesebb újdonságok közé sorolható a JavaScript **Object** konstruktor kibővítése, amelynek eredményeképp egyszerűen tudunk saját objektumokat létrehozni és másolni. A saját objektumaink tulajdonságait is egyszerűbben kezelhetjük az új **getter** és **setter** funkciókkal. Nézzünk meg egy példát új objektum létrehozására:

```
var ember = {
    nev: "Ismeretlen ember",
    lakCim: "Ismeretlen lakcím",
    születesiDatum : new Date()
};
// Eredmeny: lajos.name = "Ismeretlen ember"
var lajos = Object.create(ember);
```

A létrehozott objektumunk kibővítése sem sokkal bonyolultabb:

```
// Ember objektum kibővítése -> Férfi objektum létrehozása.
// Az újonnan létrehozott objektum "nem" tulajdonsága alapértelmezetten férfi lesz.
var ferfi = Object.create(ember);
Object.defineProperty(ferfi, "nem", { value: "férfi" });

// lajos.nem = "Férfi"
var lajos = Object.create(ferfi);
```

A JavaScript nyelvben nagy újdonságnak számító **getter** és **setter** függvények segítségével objektumorientált módon tudjuk kezelni az objektumaink tulajdonságait:

```
// Ember objektum kibővítése -> Férfi objektum létrehozása.
// Az újonnan létrehozott objektum "nem" tulajdonsága alapértelmezetten férfi lesz.
var ferfi = Object.create(ember);
Object.defineProperty(ferfi, "nem",
{
    // A tulajdonság mindig a "férfi" értéket adja vissza
    get: function () { return "férfi" },

    // Ha módosítani próbálnánk a tulajdonságot, akkor a JavaScript hibaüzenet dob
    set: function (value) { alert('Ezt a tulajdonságot nem lehet módosítani'); }
});

// ferfi objektum példányosítása
var lajos = Object.create(ferfi);
// "nem" tulajdonság felülírása. A böngésző feldobja a hibaüzenetet és nem fog történni
// semmi, a "nem" tulajdonság továbbra is "ferfi" marad
lajos.nem = "nő"
```

A fent bemutatott példán túl még számtalan hasznos objektumkezelési lehetőséget nyújt a nyelv, ám a könyv terjedelme nem engedi meg az összes újdonság bemutatását.

A dokumentum elemeinek visszakeresésére is találunk új funkciókat, amelyekkel a korábbiakhoz képest sokkal komplexebb keresési feltételeknek megfelelően találhatjuk meg a dokumentumunk bizonyos elemeit:

- **document.getElementsByClassName(argStr):** CSS osztályok alapján tudunk keresni
- **document.querySelector(argStr):** CSS kiválasztók alapján tudunk keresni (csak az első megtalált elemmel tér vissza)
- **document.querySelectorAll(argStr):** CSS kiválasztók alapján tudunk keresni (az összes megtalált elemmel tér vissza)

A JavaScript területet érintő hasznos újítások sorából érdemes még kiemelni az új tömb funkciókat is, amelyek az adatfeldolgozást hivatottak megkönnyíteni, leegyszerűsíteni (és talán oly sok éve hiányoznak a JavaScript eszköztárából). Ezeknek a metódusoknak a sorában olyan függvényeket találunk, mint például az **Array.every(fn)**, **Array.some(fn)**, **Array.forEach(fn)**, **Array.map(fn)**, **Array.filter(fn)** és **Array.reduce(fn)** függvények.

A felsorolt metódusok erősségét a gyakorlati alkalmazásuk példázza a legjobban, ezért tekintsük meg őket működés közben:

```
// Páros számok vizsgálatát végző függvény
function isEven(element) {
    return element % 2 == 0;
}
```

```
// A tömb egy elemét és indexét kiíró függvény
function printArray(element, index, array) {
    document.write("Value: " + element + ", Index: " + index + "<br />");
}

// A megadott paramétert negáló függvény
function negateArray(element) {
    return 0 - element;
}

// A tömb elemeit összefésülő függvény
function sumArray(prevElement, nextElement) {
    return prevElement + nextElement;
}

// Megnézzük, hogy a tömb összes eleme páros-e
// A visszatérési érték: false, mert van közöttük páratlan elem
sampleNumberArray.every(isEven);

// Megvizsgáljuk, hogy a tömb elemei között találunk-e párosat.
// A visszatérési érték: true, mert van közöttük páros elem
sampleNumberArray.some(isEven);

// A tömb összes elemét kiírjuk a dokumentumra
sampleNumberArray.forEach(printArray);

// A tömb összes elemét negáljuk, majd a függvény visszatérési értéke egy új tömb lesz,
// a következő elemekkel: 0, -9, -2, -8, -1, -5, -7, -3, -4
sampleNumberArray.map(negateArray);

// Megkeressük a tömb összes páros tagját és egy új tömbben visszatérünk vele,
// így tehát az eredmény: 0, 2, 8, 4
sampleNumberArray.filter(isEven);

// Tömb elemeinek összeadása. Eredmény: 39
sampleNumberArray.reduce(sumArray);
```

Szorosan a JavaScript újítások közé tartoznak az új XML feldolgozó függvények is, amelyek segítségével szöveges adatokat tudunk DOM elemekké (leegyszerűsítve XML elemekké) alakítani, és fordítva. Tehát az adattároláshoz és feldolgozáshoz immáron XML formátumú adatokat is fel tudunk használni:

```
// DOM (XML) értelmező inicializálása
var parser = new DOMParser();

// Példa adat létrehozása
var inputData = "<files><file>" +
    "<name value=\"samplePic01.png\"/>" +
    "<type value=\"image/png\"/>" +
    "<type fileSize=\"123654\"/>" +
    "</file><file>" +
    "<name value=\"M text document.txt\"/>" +
    "<type value=\"text/plain\"/>" +
    "<type fileSize=\"697\"/>" +
    "</file></files>";

// Szöveges adat konvertálása szabványos XML elemekké
var convertedData = parser.parseFromString(inputData, "text/xml");
// XML objektumból szöveges adatot konvertáló "osztály" inicializálása
var serializer = new XMLSerializer();

// XML dokumentum konvertálása szöveggé. Eredmény: az eredeti szöveges adat
var xmlString = serializer.serializeToString(element);
```

A fenti funkciók meglepte nem tűnik nagy újdonságnak, hiszen a fejlettebb programozási környezetek többsége szolgáltat megoldásokat XML feldolgozásra. Ám JavaScript környezetben ezek idáig csak valamilyen külső JavaScript keretrendszer segítségével voltak megoldhatók.

A HTML5-öt kiegészítő modulok

A fejezetben korábban már megismerkedhettünk a HTML5 leíró nyelv és a nyelvhez szorosan kapcsolódó JavaScript legizgalmasabb funkcióival és újdonságaival. A továbbiakban olyan technológiák kerülnek bemutatásra, amelyek nem a HTML5 nyelv részét képezik, ám mégis szoros kapcsolatban állnak vele. Mint ahogy látni fogjuk, ezek a modulok mind a felhasználói, mind a fejlesztői elvárások növekedésének eredményei, és egy-egy speciális felhasználási terület kiaknázására szolgálnak. Fontos megjegyezni, hogy a bemutatásra kerülő modulokon felül még számtalan más modulja is van a HTML5-nek, ám a fejezet írásakor nagy részük még olyannyira félkész állapotban van, hogy bemutatásuk értelmetlen lenne. Ezeknek az ígéretes, ám félkész moduloknak a körébe tartozik a kliens számítógép eszközeit (web kamera, mikrofon) kezelő **device** modul, a beviteli vezérlők kibővítésére szolgáló **web forms 2.0** modul, a **web sockets** modul vagy a háromdimenziós grafikákat és animációkat összefoglaló **WebGL** modul.

Az SVG modul

Az SVG modul azt a célt szolgálja, hogy a böngészőnk segítségével, külső bővítmények nélkül tudjunk megjeleníteni SVG formátumú vektorgrafikákat. Az így megjelenített SVG grafikákról dióhéjban annyit érdemes tudni, hogy a végeredmény XML formátumban leírt vonalak, görbék és egyéb kétdimenziós alakzatok halmaza. Ha közelebbről megvizsgáljuk, némi párhuzamot lelhetünk fel a HTML5 vászon eleme és az SVG között, hiszen mindkettő segítségével kétdimenziós alakzatokat rajzolhatunk a böngészőbe, amelyeket tetszőleges színnel, színátmenettel és mintával tudunk kitölteni. Az alakzatok kitöltésén túl az SVG elemeit animálhatjuk, geometriai transzformációkat hajthatunk végre rajtuk, és olyan szűrőket alkalmazhatunk az elemekre, mint pl. a Gauss elmosás (*gaussian blur*). Tekintettel arra, hogy a fejezet nem az SVG technológia bemutatására szolgál, nézzük meg egy példán keresztül, hogyan ágyazható a dokumentum forrásába egy SVG grafika:

```
<section>
<svg width="600" height="300" version="1.1" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <filter id="Gaussian_Blur">
      <feGaussianBlur in="SourceGraphic" stdDeviation="3"/>
    </filter>
  </defs>
  <rect x="20" y="20" rx="20" ry="20" width="200" height="100"
    style="fill: red; stroke: black; stroke-width: 5; opacity: 0.5; filter: url(#Gaussian_Blur)" />

  <rect x="240" y="0" width="200" height="200" style="fill: blue">
    <animate attributeType="CSS" attributeName="opacity" from="1" to="0" dur="5s"
      repeatCount="indefinite" />
  </rect>
</svg>
</section>
```

Kliens oldali adattárolás

Kliens oldali adattárolásról akkor beszélünk, amikor adott egy webes alkalmazás, és az alkalmazáshoz kapcsolódó adatokat a böngészőben tároljuk. Miközben egyre divatosabbak a felhő, számítási felhő (*cloud computing*) kifejezések, és a szakmai cikkek is arról szólnak, hogy a jövőben minden adatunk a felhőben lesz tárolva, mégis elkerülhetetlen, hogy bizonyos adatokat kliens oldalon tároljunk. És ekkor jön kapóra a HTML5 egyik újdonsága, a **Web Storage** modul.

A HTML5 megjelenése előtt is léteztek megoldások a kliens oldali adattárolásra. A legelterjedtebb, böngésző és platform független megoldásnak a süti (*cookie*) bizonyult. A sütiknek azonban több hátrányuk is van, a legnagyobb talán az, hogy igencsak korlátozott annak az adatnak a mérete, amit bennük

tárolhatunk (4 KByte). A másik hátránya pedig az, hogy minden szerverkérésnél a kérés fejlécében továbbítva van a süti, ami teljesen felesleges adatforgalmat eredményezhet bizonyos esetekben. A HTML5 Web Storage modul kétféle megoldást is kínál az adataink tárolására. Az egyik alternatíva a *Session Storage*, a másik pedig *Local Storage*. Mind a Session Storage, mind a Local Storage egy közös Storage API-ból származik, ezért a két megoldás implementálásának módja azonos (ugyanazokkal a függvényekkel és tulajdonságokkal dolgozhatunk), a különbség csupán az adatok felhasználásának módjában jelentkezik.

Mindkét tárolási megoldás esetében az adatokat kulcs–érték pár formájában lehet tárolni és visszakeresni. A kulcs nem más, mint egy szöveges objektum, az érték pedig bármilyen típusú objektum lehet, amit a böngésző támogat.

És most nézzük meg a rendelkezésre álló függvényeket: a **getItem(key)** metódussal az adott kulcshoz tartozó elemet vehetjük ki a tárolóból, a **setItem(key,value)** függvénnyel adatot tehetünk be a tárolóba, a **removeItem(key)** segítségével az adott kulcshoz tartozó elemet törölhetjük a tárolóból, a **clear()** függvénnyel pedig a tárolót üríthetjük ki.

A Session Storage

Tulajdonképpen a Session Storage objektum szolgáltat megoldást a sütik kiváltására. Működése nagyon hasonlít a sütik működéséhez, annyi különbséggel, hogy segítségével nem 4KB adatot lehet tárolni, hanem böngészőfüggően több megabájtot (általánosságban max. 5MB / domain). A másik különbség az, hogy az adat nem minden kérésnél van továbbítva a kiszolgáló felé, hanem csak akkor, ha változás történik benne, ezért a kiszolgáló terhelése lényegesen csökkenthető. Az adott domainhez tartozó Session Storage objektum csak az aktuális ablakra érvényes, azaz, ha van egy oldalunk, és még egyszer megnyitjuk egy másik ablakban, akkor az újonnan megnyitott ablak Session Storage objektuma üres lesz.

A Local Storage

A Local Storage lényegében egy speciális JavaScript objektum, ami arra szolgál, hogy olyan adatokat tároljon az adott domainhez, ami az ablak, oldal bezárása után is elérhető. Ha az adott oldalt egy másik fülön vagy böngészőben megnyitjuk, akkor a Local Storage objektum tartalma nem fog megváltozni, benne lesz, amit már korábban beletettünk. Segítségével többek között olyan nagy mennyiségű adatokat tárolhatunk kliens oldalon, mint pl. a felhasználók mailjei.

```
// Létrehoztunk két változót, az egyik a LocalStorage, a másik a SessionStorage
// tárolónak felel meg.
var localStorage = window['localStorage'];
var sessionStorage = window['sessionStorage'];

// Adat elhelyezése a LocalStorage tárolóba
localStorage.setItem("test", "LocalStorage test");

// Adat lekérése a LocalStorage tárolóból. Eredmény: "LocalStorage test"
var value = localStorage.getItem("test");

// LocalStorage tároló ürítése
localStorage.clear();

// Újból lekérjük a "test" kulcshoz tartozó adatot a tárolóból.
// Eredmény: null, hiszen kiürítettük a tárolót
value = localStorage.getItem("test");
```

A GeoLocation modul

A **GeoLocation** modul segítségével visszakaphatjuk a weboldalt megjelenítő eszköz földrajzi koordinátáit, amit később egy térképen meg tudunk jeleníteni — például Bing Maps segítségével. Mindez nagyon érdekesnek tűnhet, a modul szépséghibája csupán annyi, hogy nincs pontosan meghatározva az, hogy hogyan kell meghatározni az eszköz helyzetét. Többek között meghatározhatjuk az eszközbe épített GPS segítségével, mobil cellainformációk alapján vagy IP cím alapján is. A hiányos specifikáció miatt az eszköz

koordinátái böngészőnként eltérhetnek, és csak abban az esetben kaphatunk pontos eredményt, ha a GPS segítségével határozzuk meg az eszköz helyzetét.

```
// Megvizsgáljuk, hogy a böngésző támogatja-e a Geolocation API-t. Amennyiben igen,
// lekérjük az aktuális tartozkodási helyet. Az első függvény akkor hajtódik végre, ha
// sikerült, a második akkor, ha nem sikerült a lekérdezés
if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(onSuccess, onError);
}
function onSuccess(position) {
    // Földrajzi koordináták lekérdezése a position objektumból.
    // Eredmény: pl. "46.0763343, 18.2195188"
    var location = position.coords.latitude + ", " + position.coords.longitude;
}
function onError(error) {
    alert("Hiba történt!");
}
```

Az Offline Web Applications modul

Az **Offline Web Applications** modul segítségével azt a lehetőséget biztosíthatjuk a felhasználók számára, hogy akkor is böngészhessék a weboldalunkat (illetve elérjenek bizonyos funkciókat), ha valamilyen okból kifolyólag megszakad az internetkapcsolatuk. Lényeges áttörésnek számít, hogy fejlesztőként szabályozni tudjuk, hogy mi történjen a weboldallal akkor, ha nincs internet kapcsolat. A modul implementálása első ránézésre roppant egyszerűnek tűnik, csupán létre kell hozni egy egyszerű szöveges fájlt (*cache* fájl), ami azt írja le, hogy a weboldal mely részei legyenek gyorsítótárazva, majd az így létrehozott fájlra hivatkozni kell a HTML forráskódban. Ám a modul gyakorlati alkalmazásánál könnyű elbukni azon, hogy az alkalmazás mely részeit gyorsítótárazzuk (hiszen nyilvánvalóan a kiszolgálótól eredő összes fájlt nem célszerű), és azon is, hogy a böngésző a gyorsítótárban lévő fájlokat mikor töltsse le újból a kiszolgálóról, milyen időközönként frissítse a cache fájlt. Egyszerűen a modul gyakorlati alkalmazása gondos odafigyelést igényel. Most azonban nézzük meg, hogy hogyan néz ki egy ilyen cache fájl.

```
CACHE MANIFEST
# cache manifest file for www.sample.org
# erre az első sorra minden esetben szükség van

CACHE
# az itt felsorolt fájlok gyorsítótárazva lesznek a böngészőben
# ameddig az a "cache manifest" fájl meg nem változik,
# nem lesznek még egyszer letöltve
index.html
images/background.png
styles/default.css
scripts/common.js
scripts/offline.js

NETWORK:
# az itt felsorolt fájlokat a böngésző soha nem fogja gyorsítótárazni,
# hiszen ha akarná, se tudná ezt meg tenni
backend/upload.ashx
backend/getData.aspx

FALLBACK
# ha a megjelenítendő oldal nincs gyorsítótárazva, akkor a "Lap nem jeleníthető meg"
# üzenet helyett a böngésző az "/offline.html" fájlt fogja megjeleníteni.
# az első "/" jel jelenti azt a mintát, hogy bármely nem letölthető fájl esetében
# az "/offline.html" fájlt jelenítse meg a böngésző
/ /offline.html
```

Azután, hogy a fenti fájlt elhelyeztük a kiszolgálón, már csak hivatkozni kell rá a dokumentum forrásában.

```
<!DOCTYPE html >
<html manifest="/cache.manifest">
  <head>...</head>
  <body>...</body>
</html>
```

A Web Workers modul

A **web workers** modul lehetőséget biztosít a JavaScript kódok háttérben való futtatására: több párhuzamos szálon végezhetünk egyszerre számításigényes feladatokat, szerverhívásokat úgy, hogy eközben a felhasználói felület nem fagy le, és azt a felhasználók folyamatosan interaktívnek érzékelik. Természetesen ez a lehetőség óriási segítség a fejlesztők számára, hiszen fejlesztői szemmel megszabadulhatunk attól a teherből, hogy biztosítani kell az aktív felhasználói felületet számításigényes feladatok elvégzése közben. Felhasználói szemmel tekintve pedig kedvenc webes alkalmazásaink sokkal gyorsabbak, válaszra készek lehetnek a korábbiaknál.

A modul részletes ismertetése helyett inkább nézzük meg annak gyakorlati felhasználását! Először is szükségünk van egy különálló JavaScript fájlra, ami a külön szálon futtatandó kódrészletet tartalmazza, és magára a weboldalra, ahol megjelenítjük a művelet eredményét:

JavaScript fájl:

```
// prime.js. prímszámokat kereső eljárás, ami végtelen ideig fut, tehát normál esetben
// a böngésző lefagyna tőle és semmilyen visszajelzés nem érkezne a felhasználó fele
var n = 1;

search: while (true) {
  n += 1;
  for (var i = 2; i <= Math.sqrt(n); i += 1)
    if (n % i == 0)
      continue search;

  // prímszám megtalálva: a Worker objektum.postMessage()
  // függvényén keresztül jelezzük a felületnek
  postMessage(n);
}
```

HTML dokumentum:

```
<!DOCTYPE html>
<html>
<body>
  <output id="result">
  </output>
  <script type="text/javascript">

    var primeFinder = new Worker("prime.js");

    primeFinder.onmessage = function (event) {
      document.getElementById("result").innerHTML = "Prime: <b>"+event.data+"<b>";
    };
  </script>
</body>
</html>
```

A CSS 3 stílusleíró nyelv

Mint ahogy azt már a fejezet első része óta tudjuk, a HTML fejlesztésnek három elengedhetetlen pillére van: maga a HTML leíró nyelv, a JavaScript programozási réteg, és a dokumentumok megjelenését leíró CSS stílusleíró nyelv. A következőkben ezzel a stílusleíró nyelvvel fogunk megismerkedni.

A CSS3 legnagyobb előnye (legalábbis a böngészők szempontjából) az, hogy elődeivel ellentétben modulokból áll. Bizonyos modulok közel járnak a végleges változathoz, de vannak olyanok is, amelyek csak „ötletek”. A modularizált felépítésének köszönhetően az egyes újításokat a böngészők csomagban is meg tudják valósítani, így nem kell egyszerre implementálni a teljes szabványt. Ennek köszönhetően az érdekesebb és hasznosabb funkciók sokkal hamarabb eljuthatnak a felhasználókhoz. Nézzük meg, hogyan is épül fel ez a stílusleíró nyelv!

Alapvetően két fő alkotóeleme van: a *kiválasztók* (amelyek segítségével a dokumentum elemeinek keresésére adhatunk meg szabályokat), illetve a *stílus definíciók*, amelyeket az adott elemekre alkalmazhatunk (pl. háttérszín). A továbbiakban részletesebben is megismerkedünk a kiválasztókkal és a főbb modulokhoz tartozó stílus definíciókkal.

Kiválasztók

Mint ahogy már említettem, a kiválasztók azt a célt szolgálják, hogy a dokumentum struktúrából kiválasszuk azokat az elemeket, amelyekre a stílus szabályainkat alkalmazni fogjunk. Léteznek egyszerű kiválasztók és összetettebb kiválasztók is. Az egyszerű kiválasztók közé sorolhatjuk az osztály, azonosító, attribútum és típus kiválasztókat, az összetettebbek közé pedig a pszeudosztályokat és a pszeudelemeket. A CSS osztály kiválasztók segítségével nevekkal ellátott „osztályokba” foglalhatjuk össze a stílus definícióinkat, és később a HTML elemek definiálásakor az elemek **class** tulajdonságát felhasználva hivatkozhatunk a CSS osztályokra.

```
<head>
  <style>
    .pirosOsztaly
    {
      background-color: Red;
      color: White;
    }
  </style>
</head>
<body>
  <span class="pirosOsztaly">Hello!</span>
</body>
```

Az azonosító alapú kiválasztókkal a HTML elemek azonosítóját (**id** tulajdonság) felhasználva tudunk stílusokat alkalmazni bizonyos elemekre.

```
<head>
  <style>
    #first { background-color: Blue; }
    #second { background-color: Green; }
  </style>
</head>
<body>
  <div id="first"></div>
  <div id="second"></div>
</body>
```

A típus kiválasztók segítségével pedig a HTML elemek típusait felhasználva tudunk stílusszabályokat alkalmazni bizonyos elemekre.

```
<head>
  <style>
    h1      { font-weight: bold; }
    p       { background-color: Blue; }
    span    { background-color: Green; }
    section { background-color: Gray; }
```

```
</style>
</head>
<body>
  <section>
    <h1>Chapter title</h1>
    <p>
      <span>Chapter content</span>
    </p>
  </section>
</body>
```

Az attribútum kiválasztók segítségével már összetettebb kiválasztási szabályokat is megadhatunk a dokumentum elemeinek tulajdonságait felhasználva. A legegyszerűbb attribútum kiválasztó segítségével azokat az elemeket lehet megtalálni, amelyek rendelkeznek a megadott tulajdonsággal. De ezenfelül megvizsgálhatjuk azt is, hogy a megadott tulajdonság milyen értékkel végződik, kezdődik, valamint milyen értéket tartalmaz.

```
/* Az összes "title" tulajdonsággal rendelkező "H1" elemre érvényes */
h1[title] { font-weight:bold; }

/* Azon "H1" elemekre érvényes, amelyek "title"
   tulajdonsága "FirstChapter" */
h1[title='FirstChapter'] { font-size: 120%; }

/* Azokra az "IMG" elemekre érvényes, amelyek "contenttype"
   tulajdonsága "image"-el kezdődik */
img[contenttype^="image"] { border:solid 1px rgb(255,0,0); }

/* Azokra az "IMG" elemekre érvényes, amelyek "contenttype"
   tulajdonsága "jpeg"-el végződik */
img[contenttype$="jpeg"] { border:solid 1px rgb(0,255,0); }

/* Azokra az "IMG" elemekre érvényes, amelyek "contenttype"
   tulajdonsága tartalmazza a "png"-t */
img[contenttype*="png"] { border:solid 1px rgb(0,0,255); }
```

Az egyszerű kiválasztók után most nézzük meg a pszeudosztályokat! Segítségükkel a vezérlő elemek különböző állapotait és a dokumentum struktúrában betöltött helyzetüket vizsgálva tudunk stílusszabályokat definiálni. Például annak függvényében tudunk különböző stílust alkalmazni egy hivatkozásra, hogy a kurzor a hivatkozás felett van-e vagy sem. Vizsgálhatjuk például azt is, hogy egy **checkbox** vezérlő elem kijelölt állapotban van-e vagy sem. Az n-edik elem kiválasztókkal pedig az elemek elhelyezkedésének függvényében tudunk stílusokat definiálni. A pszeudosztályok körébe tartozik még a negáló kiválasztó, amely paraméterként egy másik kiválasztó kifejezést vár, és azokra az elemekre alkalmazza a stílust, amelyeket a paraméterként átadott kiválasztó nem talált meg.

```
/* Akkor érvényes a szabály, ha a hivatkozás felett elhalad a kurzor */
a:hover { color: Green; }

/* Azokra a szöveges beviteli vezérlőkre érvényes, amelyek állapota "disabled" */
input[type=text]:disabled { opacity: 0.5; }
/* Táblázatok páratlan sorainak formázása */
tr:nth-child(2n+1) { color: rgba(0,255,0,0.9) };
/* Ugyan az, mint az előző */
tr:nth-child(odd) { color: rgba(0,255,0,0.9) };

/* Táblázatok páros sorainak formázása */
tr:nth-child(2n+0) { color: rgba(0,255,0,0.7) };
/* Ugyan az, mint az előző */
tr:nth-child(even) { color: rgba(0,255,0,0.7) };
```

```
/* Az összes olyan beviteli vezérlő elemre érvényes, amelyek nem "checkbox"-ok */
input:not([type=checkbox]) { background-color:Red; }
```

A pszeudosztályok után nézzük meg, hogy mire szolgálnak a pszeudoelemek! A legnagyobb különbség a két kiválasztó csoport között az, hogy a pszeudoelemek olyan kiválasztók, amelyek nem a dokumentum felépítése és szerkezete alapján alkalmaznak stílusszabályokat, hanem ettől függetlenül. Segítségükkel például egy bekezdés első sorára vagy első betűjére alkalmazhatunk bizonyos stílusokat, vagy dinamikus tartalmat szűrhatunk be az elemek elé, mögé vagy köré. A dinamikus tartalomgenerálásra később még visszatérünk.

```
/* Bekezdések első sorának formázása */
p::first-line { color: Blue; }

/* Bekezdések első betűjének formázása */
p::first-letter { font-size: 200%; }
```

A Media queries modul

A **Media queries** modul segítségével függetleníthetjük weboldalunkat a megjelenítő eszköztől: ha ugyanazt a dokumentumot hagyományos számítógépen, netbookon vagy okos telefonon szeretnénk megjeleníteni, nem szükséges újraírni a dokumentum szerkezetét, mert a **media queries** modul segítségével olyan globális stílusokat hozhatunk létre, amelyek minden megjelenítő eszközre érvényesek, és az így létrehozott stílusokat kiegészíthetjük megjelenítő eszközspecifikus stílusokkal is. Tegyük fel, hogy van egy olyan felületünk, amit meg lehet tekinteni böngészőben, másrészt ki lehet nyomtatni (pl. e-számla). Ebben az esetben is segíthet a **media queries** modul: külön megírhatjuk a nyomtatási stílust és a normál megjelenítéshez tartozó stílust is. Online multimédiás szolgáltatásoknál különösen fontos lehet, hogy az adott oldalt meg lehet tekinteni hagyományos számítógépen és tévéen is: ebben az esetben is létrehozhatunk két külön stílust a két különböző megjelenítőhöz. Ám minden webfejlesztő legnagyobb rémálma mégiscsak az, hogy a felhasználók különböző felbontással tekintik meg az oldalt. Szerencsére ebben is segítségünkre lehet a modul, hiszen a felbontástól függően is más – más stílusokat alkalmazhatunk a weboldalainkon.

```
@media all and (min-width: 640px) {
    .class { background-color: green; }
}
@media screen and (max-width: 1400px) {
    .class { background-color: green; }
}
@media print and (min-resolution: 300dpi) {
    .class { background-color: green; }
}
@media tv and (scan: progressive) {
    .class { background-color: green; }
}
```

A Colors modul

A CSS3 **colors** moduja hivatott a weben használt színkezelés megreformálására, hiszen a korábban hexadecimálisan és szöveges formában történő színdefiníálás mellett „okosabb” módon is használhatunk színeket. Használhatjuk az RGB, illetve RGBA és a HSL, illetve HSLA színkezelési modelleket is, így a webes alkalmazás arculattervezését követően sokkal egyszerűbb HTML-be ágyazni a különböző grafikai elemeket.

```
.red
{
    color: rgb(255,60,30);
    color: rgba(255,60,30,0.7);
    color: hsl(0, 75%, 100%);
    color: hsla(0, 75%, 100%, 0.7);
    color: #FF3C3C;
}
```

Dinamikus tartalomgenerálás

A CSS lehetőséget nyújt arra is, hogy stílusból szűrjünk be szöveges tartalmat bizonyos elemek elé, mögé vagy az adott elem köré. Mindezt a már korábban említett pszeudoelemek és speciális kulcsszavak, illetve függvények segítségével tudjuk megtenni.

```
/* Megjegyzés szó és a megjegyzéshez tartozó azonosító beszúrása a "comment" osztályba
   tartozó bekezdések elé. */
p.comment::before {
    content: 'Megjegyzés (' attr(commentID) ' )'
}

/* Az alábbi blokk segítségével az összes "div" elem a következőképpen fog megjelenni:
   "D C B A" */
div { content: 'A' }
div::before { content: 'B'; }
div::before(2) { content: 'C'; }
div::before(3) { content: 'D'; }

/* Többszörös keret rajzolása a "div" elemeink köré */
div::outside { border: dashed }
div::outside(2) { border: dashed }
```

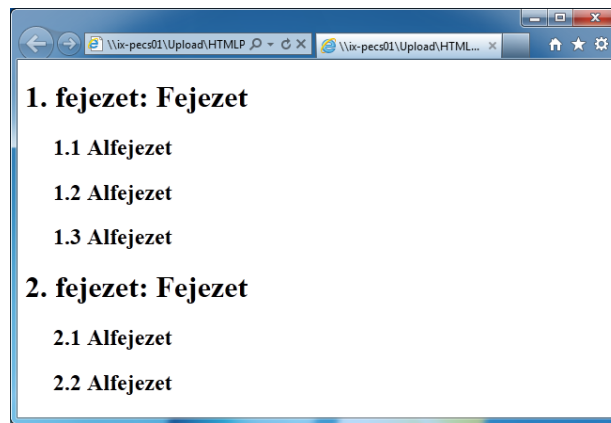
A dinamikus tartalomgenerálás elemek sorszámozására is lehetőséget nyújt. Tegyük fel, hogy van egy olyan dokumentumunk, amelyben több fejezet és alfejezet cím is található. Ebben az esetben nem szükséges kézzel sorszámozni a címeket, helyette használhatjuk a beépített számlálót és a dinamikus tartalomgenerálást.

```
/* Eredmény: 1. Bekezdés;; 1.1;1.2;1.3; 2. Bekezdés; 2.1; 2.2 */

h1 {
    /* 'section' alfejezet számláló alaphelyzetbe állítása */
    counter-increment: chapter;
    counter-reset: section;
}

H1:before {
    /* 'chapter' nevű számláló növelése minden bekezdésnél */
    content: counter(chapter) '. fejezet: ';
}

H2:before {
    /* 'section' számláló növelése */
    content: counter(chapter) "." counter(section) " ";
    counter-increment: section;
}
```



13-6 ábra: Dinamikus tartalomgenerálás

A Template layout modul (CSS sablonok)

A **template layout** modul segítségével a dokumentumunkat feloszthatjuk több részre (rácsszerűen), és a weboldal minden elemét ezekben az egységekben helyezhetjük el. Ennek a megközelítésnek az a nagy előnye, hogy nagyobb és összetettebb megjelenésű dokumentumok tervezésekor a végeredmény átláthatóbb és szemléletesebb lesz.

```
body {
  height: 100%;
  display: "a . b . c" /80px
          ". . . ." /5px
          "d . e . f"
          ". . . ." /5px
          "g . h . i" /50px
          80px 5px * 5px 80px
}
#logo { position: a}
#motto { position: b}
#date { position: c}
#main { position: e}
#adv { position: f}
#copy { position: g}
#about { position: h}
```

A fenti példában a dokumentumunkat felosztottuk három sorra és három oszlopra. Az első sor magassága 80 pixel, a második és a negyedik sor elválasztóként funkcionál, és a magasságuk 10 pixel. A középső sor magassága változó, a sablonba helyezett tartalomtól függ. Az utolsó sor szintén fix magasságú. Az oszlopokat tekintve pedig az első és utolsó oszlop szélessége statikusan 80 pixel. Az oszlopok között szintén találunk két elválasztóként funkcionáló sablont. A középső oszlop szélessége pedig szintén a felbontás és a tartalom függvényében változik. Azután, hogy definiáltuk a sablonokat, a dokumentum egyes részeit ezeken a sablonokon belül helyezzük el.

A Background and borders modul

Ez a modul tartalmazza többek között a már nagyon sok éve hiányolt lekerekített sarkok és árnyékolt elemek használatához szükséges eszközöket. Újdonságnak számít, és szintén ebbe a modulba tartozik az, hogy elem köré nemcsak egyszerű körvonalat rajzolhatunk, hanem többszínű, mintával kitöltött kereteket is. Ez utóbbi főleg képszerkesztő és képkezelő alkalmazásokban lehet igen hasznos.

```
.box
{
    /* lekerekített sarkok rajzolása */
    border-radius: 15px;
    border-top-left-radius: 10px 5px;
    /* árnyék rajzolása */
    box-shadow: 5px 5px 2px rgba(0,0,0,0.5);

    /* belső árnyék rajzolása */
    box-shadow: inset -5px -5px 5px rgba(0,0,0,0.5);
}
```

A Fonts modul

A CSS3 betűtípusok modulja foglalja össze a betűtípusok stílusának leírásához használható olyan stílus definíciókat, mint például a betűtípus, betűméret és betűstílus. Szintén ebben a modulban találhatjuk meg a CSS3 egyik legérdekesebb új képességét is: a beágyazható betűtípusokat. A beágyazható betűtípusok segítségével immáron saját betűtípusokat szűrhatunk be a dokumentumainkba, és nem kell azon aggódni, hogy a kliens számítógépen az telepítve van-e vagy sem.

```
/* Egyéni betűtípus importálása */
@font-face {
    font-family: DymaxionScript;
    src: url("DymaxionScript.woff");
}

/* Az importál betűtípus felhasználása */
p { font-family: DymaxionScript; }
```

A transzformációs modul (2D és 3D)

A CSS transzformációk segítségével vászon és JavaScript segítségével nélkül tudunk a dokumentumunk elemeire különböző geometriai transzformációkat alkalmazni. Ez az új szemléletmód nagymértékben kibővíti a webes arculattervezés lehetőségeit, hiszen a **media queries** modullal közösen nemcsak függetleníteni tudjuk a megjelenítőtől az oldal megjelenését, hanem azt teljes mértékben meg tudjuk változtatni transzformációk segítségével. A JavaScripttel együtt pedig nagyon látványos animációkat készíthetünk vászon és egyéb elemek felhasználása nélkül, csupán pár sornyi forráskóddal.

```
/* 2D Transzformációk */
.scale      { transform: scale(1,0.5);      }
.rotate     { transform: rotate(45deg);     }
.translate  { transform: translate(10px, 20px); }

/* 3D Transzformációk */
.scale3D    { transform: scale3d(1.5,1.5,2.0);    }
.rotate3D   { transform: rotate3d(0,0,1, 45deg);   }
.translate3D { transform: translate3d(0px, 0px, 37px); }
```

Az animációs modul

Az animációs modul segítségével egyszerűbb animációkat készíthetünk pusztán a stílusleíró nyelv használatával. Természetesen, mivel ebben az esetben szó sincs programozásról, így összetettebb animációk készítésére ez alkalmatlan. A modul segítségével mégis látványosan feldobhatjuk a weboldalunk megjelenését.


```
div:hover {
  animation-name: 'diagonal-slide';
  animation-duration: 5s;
  animation-iteration-count: 10;
}
@keyframes 'diagonal-slide' {
  from { left: 0; top: 0; }
  to { left: 100px; top: 100px; }
}
```

Az áttűnés modul

A CSS3 áttűnés modul segítségével az animációs modulhoz hasonlóan, JavaScript felhasználása nélkül tudunk CSS tulajdonságokat animálni. A különbség a két modul között annyi, hogy ebben az esetben az animáció akkor fog lejátszódni, ha az adott tulajdonság értéke megváltozik.

```
.ImageSlider img
{
  position: absolute; width:130px; height:170px;
  /* "opacity" tulajdonság változása esetén 1mp alatt vált
  az eredeti érték az új értékre */
  transition: opacity 1s ease-in-out;
}

/* Ha a kurzor a kép fölé navigáljuk, megváltoztatjuk a kép "opacity" tulajdonságát.
Ezzel pedig kiváltjuk az áttűnés hatást */
.ImageSlider img:hover { opacity: 0; }

...

<div class="ImageSlider">
  
  
</div>
```

A fenti példában létrehoztunk két képet, amelyeket a display tulajdonságuk révén a böngésző egymásra helyez, és így egy képként jeleníti meg. A felső kép fölé mozgatva az egeret a kép egy másodperc alatt elhalványul, így előtűnik az alatta levő kép. Mindez JavaScript felhasználása nélkül.

Összefoglalás

Ebben a fejezetben megismerkedhetünk egy olyan technológiával, aminek alapszintű ismerete elengedhetetlen minden webes technológia, így a Silverlight alkalmazásához is. Azonfelül, hogy nélkülözhetetlen eleme a webes világnak, a HTML5 széles körű megoldásokat kínál számos Silverlight funkció kiváltására.

Láthattuk, hogy mindösszesen pár sornyi HTML, CSS és JavaScript kombinálásával nagyon egyszerűen, látványosan és szemléletesen készíthetünk animációkat, az üzleti alkalmazásokhoz elengedhetetlen űrlapokat és listákat, és még számtalan funkciót, amelyeknek csak a képzeletünk szab határt.

Egy jó fejlesztő a megvalósítandó feladatnak megfelelően választ fejlesztői környezetet és nyelvet: a Silverlight technológiát akkor veszi elő, amikor az alkalmazásban a HTML5 technológiák segítségével csak túl bonyolultnak megoldható funkciót kell létrehozni. Természetesen komplex üzleti alkalmazások fejlesztéséhez célszerű a Silverlight technológiát alkalmazni, mivel többek között arra is lehetőséget biztosít, hogy többretegű, átlátható és könnyen bővíthető alkalmazás-struktúrát készítsünk.