Gyors alkalmazásfejlesztés GNOME alá Mono-val

Fejlesszünk GNOME környezetben a nyílt forrású .NET keretrendszerrel, a Mono-val.

A *Mono* a *.NET* fejlesztői környezet hatékony, nyílt forrású megvalósítása. Alkotóelemei egy közös nyelvi infrastruktúra (*CLI*, azaz *Common Language Infrastructure*) virtuális gép, egy *C#* fordító és számos osztálykönyvtár. *C#* nyelv és futtató környezet megvalósítása megfelel az *ECMA 334* és *335* szabványoknak.
A *Mono* – ami egyébként spanyolul majmot jelent – különböző osztálykönyvtárakat biztosít, többek között a *.NET* keretrendszer fejlesztői csomagjának egy nyílt forráskódú megvalósítását. Ebben a cikkben a *Mono* egyik leghasznosabb szolgáltatását vesszük górcső alá: a *GNOME* támogatását *Gtk#* formájában.
A *Gtk#* egy *.NET* nyelvi összekötő a *Gtk+* eszközkészlethez és számos más *GNOME* könyvtárhoz. Több, mint egyszerű csomagoló, hatékony platform grafikus felületű alkalmazások fejlesztéséhez *GNOME* környezetben. A *GTK#* nyelvi kötései kitűnő objektum-orientált alapot biztosítanak *C#* stílusú tervezéshez, egyszerűvé, mégis rugalmassá és hatékonnyá téve a *GNOME* fejlesztést.
A cikkben egy egyszerű *C#* alkalmazás szerkezetét tanulmányozzuk. A legegyszerűbb *„Hello, World!"* alkalmazással kezdjük, végül egy egyszerű *Wikipedia* keresőprogrammal fejezzük be. A függőségek mindössze a *Mono* és a *Gtk#* lesznek. Ezek a csomagok a legtöbb terjesztéshez elérhetők – lásd a megfelelő online forrásokat.

Hello, World!

Kezdjük a lehető legegyszerűbb *Mono* alkalmazással, mely a *„Hello, World!"* karakterláncot írja a képernyőre:

```
using System;
class first {
    public static void Main
    ➥(string[] args)
    {
        Console.WriteLine
        ➥("Hello, World!");
    }
}
```

Nyissuk meg kedvenc szerkesztőprogramunkat, másoljuk be ezt a kódot és mentsük *first.cs* néven. Ezután a következő paranccsal lefordíthatjuk a programot egy futtatható állománnyá:

```
$ mcs first.cs
```

Végül a következő paranccsal futtathatjuk:

```
$ mono first.exe
Hello, World!
```

Ez az alkalmazás tartalmazza az első osztályt. Minden programban szükség van egy belépési pontra, egy kezdőfüggvényre az osztályban, ahonnan a *Mono* futtatókörnyezet elkezdheti a program futtatását. Ez a függvény a Main, ahogyan a *C*-ben és a *C++*-ban is. A függvény prototípusa az alábbi:

```
public static void Main
➥(string[] args)
```

Programunk Main függvényében meghívunk egy egyszerű, *WriteLine* nevű függvényt, amely a `Console` osztályban található. A függvény a `printf()`-hez hasonlóan szöveget ír a kimenetre. Arra is használhatjuk, hogy változók értékét kiírassuk vele:

```
int x = 5;
String s = "wolf";
Console.WriteLine ("x={0}
➥s={1}", x, s);
```

Eredményül ezt kapjuk:

```
x=5 s=wolf
```

A Hello, World! színesben

A *„Hello, World!"*-öt természetesen nem csak a konzolon jeleníthetjük meg, *Gtk#*-pal egy egyszerű *GUI* párbeszédablakot is készíthetünk neki:

```
using System;
using Gtk;
class Two {
    static void WindowDelete
    ➥(object o,
    ➥DeleteEventArgs args)
    {
        Application.Quit ();
    }
    static void InitGui ()
    {
        Window w = new Window
        ➥("My Window");
        HBox h = new HBox ();
        h.BorderWidth = 6;
        h.Spacing = 6;
        w.Add (h);
        VBox v = new VBox ();
        v.Spacing = 6;
        h.PackStart (v, false,
        ➥false, 0);
        Label l = new Label
        ➥("Hello, World!");
        l.Xalign = 0;
        v.PackStart (l, true,
        ➥true, 0);
        w.DeleteEvent +=
        ➥WindowDelete;
        w.ShowAll ();
    }
    public static void Main
    ➥(string[] args)
    {
        Application.Init ();
        InitGui ();
```

```
        Application.Run ();
    }
}
```

Csakúgy, mint az előbb, írjuk be a kódot kedvenc szerkesztőnkbe, és mentsük *two.cs* néven. A program fordításához meg kell mondanunk a *Mono* fordítónak, hogy a *Gtk#* könyvtárat szeretnénk használni:

```
$ mcs two.cs -pkg:gtk-sharp
```

Futtatni ugyanúgy kell, mint az előbb:

```
$ mono two.exe
```

Az alkalmazás létrehoz egy kis ablakot, amelynek címe *My Window* lesz és a *„Hello, World!"* üzenetet jeleníti meg benne (*1. ábra*). Az ablak egy `GtkWindow`, a címke pedig egy `GtkLabel`. A *Gtk* az ablakokat dobozokra osztja. A dobozok láthatatlan grafikus elemek, kizárólag az elrendezés miatt léteznek, vagyis, hogy más elemeket tartalmazzanak. A elemek elrendezését az ablakon belül a elemek dobozbeli elrendezése határozza meg. Bár *Gtk*-ban az elemek táblázatokkal is elrendezhetők, a legtöbb programozó dobozokat használ azok rugalmassága és hatékonysága miatt. Ráadásul, ha egyszer alaposan megismerjük a dobozokat, utána már egyáltalán nem nehéz használni őket.
A *Gtk*-ban két doboztípus van: a függőleges és a vízszintes. A `vboxnak` nevezett függőleges doboz az elemek függőleges elrendezését határozza meg – oszlopokba rendezi őket. A `hbox` vízszintes doboz az elemek vízszintes elhelyezkedéséért felel, sorokba rendezi őket. Az elemeket dobozokba tesszük, azokat újabb dobozokba, amelyeket végül az ablakokhoz adunk.
Új `hbox`-ot így hozunk létre:

```
HBox h = new HBox ();
```

Új `vbox`-ot pedig így:

```
VBox v = new VBox ();
```

A dobozokat jelölő új objektumoknak különféle tulajdonságaik vannak, melyek beállításával a doboz kinézete (look and feel) szabályozható.
A következő példában beállítjuk a `hbox` két tulajdonságát:

```
h.BorderWidth = 6;
h.Spacing = 6;
```

Itt a `hbox` körüli szegélyt és térközt határoztuk meg, mindegyiket hat képpontra, így a létrejött rács térköze 12 képpont lesz. A *GNOME HIG* (*Human Interface Guideline*) esztétikai okokból és az egységesség miatt ehhez hasonlóan 12 képpont távolságot ír elő az elemek között – így a példában szereplő 6+6 képpont tökéletes.
Dobozt az ablakhoz úgy adhatunk, hogy az ablak `Add()` tagfüggvényének átadjuk a dobozt:

```
w.Add (h);
```

Elemet dobozhoz pedig a doboz `PackStart()` tagfüggvényével adhatunk:

```
public void PackStart (Widget
➥child,
            bool expand,
            bool fill,
            uint padding)
```

Példánkban ez így néz ki:

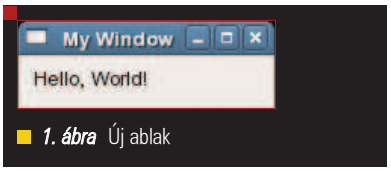
```
v.PackStart (l, true, true, 0);
```

Ez a függvényhívás a címkénket a `vbox`-ba teszi. Ha az `expand` (kiterjeszt) paraméter értéke `true`, a gyermekelem a doboz összes rendelkezésre álló területét kitölti. Ha a `fill` (kitölt) paraméter `true`, az elem a megjelenítésre felhasználja egész területét; ha `false`, akkor a felesleges területet üres lesz. A `padding` (kitöltés) paraméterrel az elem köré további térközt adhatunk a dobozon belül, más, eddig megadott térközön túl.
Alkalmazásunk futtathatóvá tétele három egyszerű lépésből áll:

```
Application.Init ();
InitGui ();
Application.Run ();
```

Az `Application.Init()` beállítja a *Gtk#*-ot és az alkalmazás grafikus felhasználói felületét. Ennek a *Gtk#* alkalmazások által meghívott első függvények között kell lennie. Ezután a program beállítja a *GUI*-ját, létrehozza és elrendezi a grafikus elemeket, megjeleníti a kezdő ablakokat és más *UI* elemeket. Ebben a programban ezt az `InitGui()` függvénnyel hajtjuk végre. Ha minden készen van, a program meghívja az `Application.Run()` függvényt, s indulhat a móka. Főablakunk felbukkan, mert az `Init.Gui()` tagfüggvényben erre utasítottuk:

```
w.ShowAll ();
```



1. ábra Új ablak

Ez megjeleníti az ablakot, benne az összes grafikus elemmel. Tehát, ha egyszer az alkalmazás meghívta az `Application.Run()`-t, felhasználói felületi elemeink megjelennek.
Futás közben a program válaszait az elemek viselkedése határozza meg. Vannak olyan elemek, amelyek előre meghatározott módon működnek, ilyenkor a programozónak nem kell saját kódot írnia. Gyakoribb azonban, hogy a programozó maga szeretné kezelni az eseményeket. Ehhez egy eseménykezelőt kell írnia, felhasználva a *C#* eseménykezelését, amelyen a *Gtk#* felhasználói beavatkozásra adott reakciói alapulnak.
Utolsó példánkban is egy ilyen eseménykezelő szerepel. Célunk az, hogy alkalmazásunk futása befejeződjön, amikor a felhasználó a főablak bezárás gombjára kattint. Kezelnünk kell tehát az ehhez kapcsolódó eseményt, a `DeleteEvent`-et, ehhez írunk egy eseménykezelőt:

```
static void WindowDelete
➥(object o, DeleteEventArgs
➥args)
{
    Application.Quit ();
}
```

Majd eseménykezelőként az ablakhoz adjuk:

```
w.DeleteEvent += WindowDelete;
```

Az `Application.Quit()` függvény hatására a *Gtk#* megsemmisíti a felhasználói felületet, bezárja és leállítja az alkalmazást. Következésképpen, ha a felhasználó rákattint a főablak bezárás gombjára, alkalmazásunk szépen befejeződik.

Egy jobb példa

Itt az ideje, hogy egy összetettebb, sőt, szinte már hasznos programocskát hozzunk létre: egy eszközt a *Wikipediában* való keresésre. Szögezzük le az elején, hogy semmi különlegeset nem csinálunk, de jól fogunk szórakozni. Létrehozunk egy

2. ábra A program grafikus felülete

egyszerű ablakot benne egy szövegbevitelre alkalmas elemmel. A felhasználó ide begépelheti a keresendő kifejezést, és egy gombra kattinthat (*2. ábra*). Az alkalmazás elindítja a felhasználó webböngészőjét és végrehajtja a keresést a ***Wikipediában***.
És ehhez csak néhány sor kódra van szükség, beleértve a ***GUI*** létrehozását és a böngészőbeli keresést is.
Az új programhoz a legutóbbi példát vesszük alapul, majd új elemeket és funkciókat adunk hozzá. A ***GTK#***-nak hála, nincs is sok új kódra szükség. Íme:

```
using System;
using Gtk;
class Example {
    public static Entry
    ➥search_entry;
    public static void
    ➥ButtonClicked (object o,
    ➥EventArgs args)
    {
        String s = "http://
➥en.wikipedia.org/wiki/Special:
➥Search?search=";
        s += search_entry.Text;
        s += "&go=Go";
        Gnome.Url.Show (s);
    }
    static void WindowDelete
➥(object o, DeleteEventArgs args)
    {
        Application.Quit ();
    }
    static void InitGui ()
    {
        Window w = new Window
➥("Wikipedia Search");
        HBox h = new HBox ();
        h.BorderWidth = 6;
        h.Spacing = 6;
        w.Add (h);
        VBox v = new VBox ();
        v.Spacing = 6;
        h.PackStart (v, false,
➥false, 0);
        Label l = new Label
➥("_Search:");
        l.Xalign = 0;
        v.PackStart (l, true,
➥false, 0);
        v = new VBox ();
        v.Spacing = 6;
        h.PackStart (v, true,
➥true, 0);
        search_entry = new Entry
➥();
        search_entry.Activates
➥Default = true;
        l.MnemonicWidget =
➥search_entry;
        v.PackStart
➥(search_entry, true, true,
➥0);
        v = new VBox ();
        v.Spacing = 6;
        h.PackStart (v, true,
➥true, 0);
        Button b = new Button
➥("Search");
        b.CanDefault = true;
        w.Default = b;
        v.PackStart (b, true,
➥true, 0);
        b.Clicked +=
➥ButtonClicked;
        w.DeleteEvent +=
➥WindowDelete;
        w.ShowAll ();
    }
    public static void Main
➥(string[] args)
    {
        Application.Init ();
        InitGui ();
        Application.Run ();
    }
}
```

Meg kell adnunk egy új, ***gnome-sharp*** nevű szerelvényt a fordítónk parancssorában. Ha programunkat ***three.cs***-nek neveztük el, ezt a következő módon tehetjük meg:

```
$ mcs three.cs -pkg:gtk-sharp
➥-pkg:gnome-sharp
```

És így futtatjuk:

```
$ mono three.exe
```

E harmadik és egyben utolsó programunk tartalmaz néhány új elemet – gombokat, címkéket és szövegmezőket –, de ugyanolyan egyszerű szerkezetű, mint az előző két példa. Ha visszafelé dolgozunk az utolsó doboztól kifelé az elsőig, már meg is van az elemek elrendezése anélkül, hogy akár egy képernyőképet is láttunk volna.
A másik nagy különbség egy új esemény, a `Clicked`. Meghatározunk egy függvényt és átadjuk eseménykezelőként:

```
b.Clicked += ButtonClicked;
```

Létrehoztuk egy nyilvános szövegbeviteli mezőt (***search_entry***), így, amikor a felhasználó megnyomja a gombot, az új eseménykezelőnk átveheti ennek a tartalmát a `search_entry.Text`-ből.
A `ButtonClicked` eseményben átvesszük a keresési kifejezést, létrehozzuk a kereséshez szükséges ***Wikipedia URL***-t, és a `Gnome.Url.Show()` függvénnyel megnyitjuk a felhasználó alapértelmezett webböngészőjét (ez egy globális ***GNOME*** beállítás), majd megnyitjuk vele az ***URL***-t.

Végszó

Nagyszerű, hogy ennyi mindent megvalósíthatunk ilyen kevés kóddal, az egészben a legszebb pedig az, hogy mindezt egy ***C*** alapú, s nem valamilyen parancsnyelven. A ***C#*** megőrzi a ***C*** hatékonyságát és teljesítményét, és ezzel együtt hatékony objektum-orientált programozási szerkezeteket is biztosít. Hadd legyek őszinte. ***C*** programozó vagyok, napjaim nagy részét kernelfejlesztéssel töltöm. A magasabb szintű nyelvek nem éppen a kedvenceim. Nem rajongok a ***C++***-ért és a ***Java***-ért sem. Mégis, mostanában, a ***C#***-pal dolgozom – például a Beagle keresőn – és teljesen lenyűgöz. A ***C#*** egy elegáns, jól megtervezett nyelv. ***Mono***-ban található szabad és nyílt ***C#*** fordító, futtatókörnyezet és számtalan könyvtár, és ráadásul élénk nyílt forrású közösség dolgozik rajta. Kitűnően alkalmas ***GNOME*** fejlesztésre, de sok más területen is jól alkalmazható.

Linux Journal 2006., 143. szám

Robert Love a Novell Ximian Desktop csoport vezető kernelfejlesztője, valamint a Linux Kernel Development (SAMS 2005) c. könyv szerzője. (A könyv második kiadása is megjelent már.) A Floridai Egyetemen szerzett diplomát CS-ből (computer science) és matematikából.

A CIKK FORRÁSAI

➲ www.linuxjournal.com/article/8750