

2. POGLAVLJE

Arhitektura .NET-a

U ovom poglavlju:

- Što je .NET Framework
- Od čega se sastoji .NET Framework
- Common Language Runtime
- Microsoft Intermediate Language
- Biblioteke klasa
- Metapodaci, *assembly* i manifesti

Nemojte se dati zastrašiti naslovom ovog poglavlja – riječ “arhitektura” u informatičkom smislu zvuči kompliciranije nego što zaista jest, a razumijevanje arhitekture .NET-a uvelike će vam olakšati napredovanje kroz knjigu i budući programerski život. Dobar dio poglavlja što slijede djelomično ili potpuno se naslanjaju na stvari koje će biti objašnjene na sljedećih nekoliko stranica, pa je uputno da ovo poglavlje “odradite” s maksimalnom pozornošću, koliko god vas vuklo da ga preskočite. Mi vam zauzvrat obećavamo da ćemo biti maksimalno jednostavni te da nećemo ići u nepotrebne detalje. Dogovoreno?

Iako se pod naslov “Arhitektura .NET-a” može ugurati zaista svašta, poglavlje će prvenstveno biti orijentirano na središnji dio .NET platforme poznat pod nazivom .NET Framework i seciranje njegove glavne komponente – Common Language Runtimea.

I. DIO: .NET IZNUTRA

.NET Framework

Najjednostavnije rečeno, .NET Framework je dio koji nadograđuje mogućnosti samog OS-a. Radi se o posebnoj infrastrukturi koja programerima nudi gotova rješenja i funkcionalnosti da bi ubrzala i pojednostavila razvoj aplikacija svih vrsta i oblika. Naravno, kako programer za vrijeme razvoja koristi ono što mu .NET nudi, logično je da će i za izvršavanje te aplikacije biti potreban .NET Framework. Stoga treba zaključiti da je .NET Framework infrastruktura nužna i za razvoj i za izvršavanje aplikacija koje koriste .NET tehnologiju.

Da tehnologija bude što prije prihvaćena i korištena, bilo je potrebno omogućiti korištenje .NET Frameworka što većem broju korisnika. Kako je nerealno očekivati da će svi odmah prijeći na Windowsa 2003 Server (s kojima, u trenutku pisanja jedinima, .NET Framework dolazi "u kutiji"), potrebno je bilo napraviti distributivnu inačicu .NET Frameworka koja se može instalirati i na starije operativne sustave.

Instalacija .NET Frameworka

Nema sumnje da će s vremenom svi operativni sustavi koji dolaze iz Microsoftovih radionica uključivati .NET Framework ili se čak i sami u potpunosti oslanjati na tu infrastrukturu. Sve inačice Windowsa izašle prije objave konačne verzije .NET Frameworka (što obuhvaća i Windows XP) trebaju se ručno nadograditi. To možete učiniti pomoću instalacijske datoteke preuzete direktno s Microsoftovih web-stranica (<http://www.microsoft.com/net>) ili preko servisa Windows Update (<http://windowsupdate.microsoft.com>). Ipak, kako ste čitanjem ove knjige na putu prema programerskim vodama, vjerojatno ste instalaciju ovog dodatka već obavili ili ćete to napraviti u paketu s instalacijom Visual Studia .NET ili nekim drugim komadom softvera koji zahtijeva instaliran .NET Framework.

Slika 2-1:
U inačicama Windowsa u kojima je potrebno doinstalirati .NET Framework, njegovo postojanje možete provjeriti u Control Panelu



2. POGLAVLJE: ARHITEKTURA .NET-A

.NET Framework, kao i svaki drugi program, s vremenom se razvija i raste te dobiva nove brojčane oznake. U trenutku pisanja knjige aktualna inačica .NET Frameworka je 1.1 i nužna je za rad s Visual Studiom .NET 2003. Oko kompatibilnosti ne morate brinuti – na istom računalu može koegzistirati više različitih inačica.



U teoriji, .NET Framework bi se mogao staviti na računalo s bilo kojim operativnim sustavom. Podrška za Windowsove operativne sustave (od Windowsa 98 nadalje) razvijena je od samog Microsofta, dok se to za ostale platforme očekuje od drugih kompanija ili organizacija.

Prva se tog posla primila kompanija Ximian (<http://www.ximian.com>) pokrenuvši projekt nazvan Mono. Cilj projekta je izraditi *open-source* implementaciju .NET Frameworka koja bi radila na Linuxu i drugim *unixoidnim* operativnim sustavima. Kako im ide i što su sve dosada napravili možete pratiti na stranicama <http://www.go-mono.com>. Projekt je vrijedan pohvale, no ostaje da se vidi hoće li Microsoftova inicijativa zaista zaživjeti na drugim platformama.

Od čega se sastoji .NET Framework?

Kako je .NET Framework osnova .NET-a, sasvim je logično da će veći dio knjige govoriti o stvari-ma koje su dio .NET Frameworka. Ipak, prije nego što krenemo na detaljnu razradu, treba sagledati kompletnu sliku i vidjeti na koji način su dijelovi međusobno povezani.

Najvažnija sastavnica .NET Frameworka zove se Common Language Runtime (naravno, nikada je u javnosti nemojte tako nazivati – uvijek koristite skraćeni oblik CLR, čitan prema engleskim pravilima sricanja). Usporedimo li .NET s računalom, CLR bi zauzeo ulogu procesora – čipa koji upravlja svime što se u računalu zbiva.

Dakako, CLR nije čip već softverski sustav u kojem se kôd pokreće i izvršava. Kada pokrenete program koji je pisan za .NET platformu, CLR ga učitava i pokreće u sebi kako bi mu osigurao stabilnost i sigurnost. Instrukcije u programu se u realnom vremenu (dakle u trenutku kada je program pokrenut) prevode u izvorni mašinski kôd (*engl. native machine code*) koji razumije računalo. Za taj je posao zadužen JIT-kompajler, što je skraćenica za *just in time*. Kako se u većini slučajeva radi o x86 kompatibilnim računalima, tako je i kôd koji JIT-kompajler generira onaj koji razumiju takva računala. Upravo zbog prevođenja u izvorni mašinski kôd računala postoji mogućnost da .NET Framework bude prenesen i na druge operativne sustave koji ne dolaze iz Microsoftovih radionica.

Dobro ste zaključili da takav način rada osjetno usporava izvršavanje aplikacije. Kompiliranje kôda, ma koliko brzo bilo, ipak zahtijeva određeno vrijeme, što će krajnji korisnik doživjeti kao sporo učitavanje aplikacije. Stoga se kompiliranje vrši samo jednom, a njegov rezultat sprema kako bi

I. DIO: .NET IZNUTRA

se kasnije mogao koristiti bez ponovnog prolaska kroz proces kompiliranja. Spremljeni kôd ponovo se kompilira tek ako se nešto u aplikaciji promijeni.



Kôd koji se izvodi unutar CLR-a nazivamo upravljani kôd (engl. *managed code*). Oni napredniji, kada ih na to natjeraju prilike, mogu pisati i/ili koristiti neupravljani kôd (engl. *unmanaged code*) koji neće biti izvršavan unutar CLR okruženja. Neupravljanim kôdom, dakle, zovemo i sve “stare” module, poput COM-komponenti.

Kao što je poznato, aplikacije za .NET platformu možete pisati u raznim programskim jezicima, gotovo svim poznatijim. CLR, međutim, ne poznaje niti jedan taj jezik – njemu naredbe dolaze isključivo u jeziku koji se zove Microsoft Intermediate Language (skraćeno MSIL, IL, a u starijoj literaturi CIL) koji je temeljen na pravilima koja se nazivaju Common Language Specification (CLS). Zaključak je jasan – mora postojati kompajler koji će programski jezik u kojem čovjek programira prevesti u MSIL kako bi ga razumio CLR.

Takvi kompajleri nazivaju se IL-kompajleri i postoje za velik broj jezika. Microsoft je napisao kompajlere za pet jezika: C#, J#, C++, Visual Basic i JScript, a ostali proizvođači softvera potrudili

Kako izgleda taj MSIL?

Za one koji jednostavno moraju vidjeti kako izgleda MSIL, evo kratkog primjera:

```
ldc.i4.4
stloc.0
ldc.i4.5
stloc.1
ldloc.0
ldloc.1
add
stloc.2
```

U prvom redu uzimamo brojku četiri i u drugom je spremamo u varijablu broj 0. U sljedeća dva reda na isti način varijabli broj 1 pridružujemo

vrijednost 5. Zatim uzimamo te dvije varijable instrukcijama *ldloc* i zbrajamo ih. Dobiveni zbroj spremamo u varijablu označenu brojem 2.

Ukoliko imate dovoljno vremena i volje za ovakvo programiranje, ili jednostavno želite ostaviti IL-kompajlere bez posla, samo izvolite. Mi ćemo se radije držati klasičnih, čovjeku razumljivijih jezika kao što je C#, u kojem gornji primjer izgleda višestruko jednostavnije:

```
int a = 4;
int b = 5;
int c = a + b;
```

2. POGLAVLJE: ARHITEKTURA .NET-A

su se oko brojnih drugih, uključujući Perl, Python, Cobol i Eiffel. Teoretski, koji god jezik koristili moći ćete napisati jednako dobre i brze aplikacije. Ipak, izaberete li neki “egzotičan” jezik, u praksi ćete imati prilično problema oko razumijevanja primjera, kako u dokumentaciji tako i na brojnim web-stranicama i grupama na Usenetu.

Iako vam se na prvi pogled MSIL može činiti kao nepotreban korak, on je direktno zaslužan za velik dio komfora koji pruža programiranje u .NET okruženju. Zahvaljujući činjenici da se sav kôd prevodi u unificirani oblik, moguće su stvari o kojima dosada nismo niti sanjali, poput nasljeđivanja klasa bez obzira na to u kojem jeziku su napisane ili *debugiranje* kompletnog rješenja bez obzira na to što su njegove komponente pisane u različitim jezicima (o čemu će biti više govora u drugom dijelu knjige).

Mogućnosti koje CLR nudi izuzetne su, no same po sebi nisu dovoljno jednostavne i upotrebljive iz ljudskog aspekta. Pisanje programa u takvom okruženju bilo bi srednjovjekovno mučenje (s tim da u informatici srednjim vijekom smatramo sedamdesete godine dvadesetog stoljeća). Stoga u .NET Frameworku postoje setovi klasa, gotovih funkcionalnosti, koje omogućavaju jednostavno i brzo iskorištavanje mogućnosti koje nudi CLR i mnogih često upotrebljivanih radnji.



Slika 2-2:
Shematski prikaz
dijelova .NET
Frameworka

Prva skupina klasa zove se bazna biblioteka klasa (engl. *Base Class Library*, skraćeno BCL) i sadrži osnovne funkcionalnosti koje koristimo u programiranju. Primjerice, trebate li u svoju aplikaciju uključiti funkcije za transformaciju teksta, mrežnu komunikaciju, provjeravanje sigurnosnih prava ili hvatanje unosa s tipkovnice, koristit ćete funkcionalnosti koje se nalaze u ovoj biblioteci.

I. DIO: .NET IZNUTRA

Svojevrсна nadogradnja osnovne biblioteke je ona koja sadrži set klasa zaduženih za suradnju s bazama podataka (ADO.NET) i XML-om. One nam omogućavaju povezivanje aplikacija s bazama podataka (poput SQL Servera), kao i manipulaciju podacima u XML dokumentima.

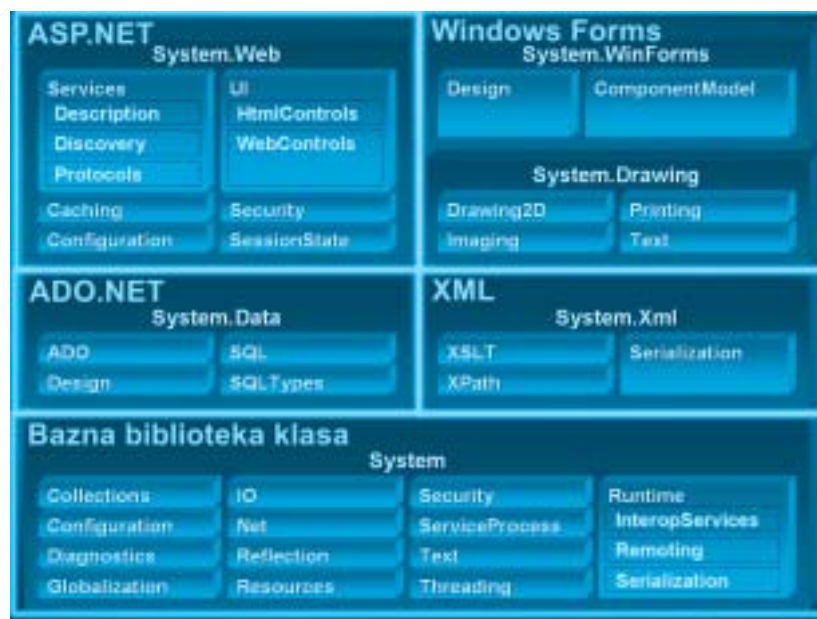
Dodatnih klasa ima još. Ukoliko se odlučite za razvoj klasične Windows aplikacije, koristit ćete klase okupljene pod imenom Windows Forms koje pružaju funkcionalnosti poput kreiranja prozora, izbornika, natpisa, polja za unos i sličnih objekata koje vidamo u klasičnim Windows aplikacijama. Naravno, klase se ne zaustavljaju samo na vizualnim detaljima, već omogućavaju upravljanje i kontrolu brojnih drugih “nevidljivih” sistemskih parametara.

S druge strane, skup klasa nazvan ASP.NET omogućit će vam razvoj dva tipa aplikacija – Web Forms i Web Services. Web Forms predstavljaju nadogradnju ili, bolje rečeno, zamjenu za serverско skriptiranje na web-stranicama (za što se prije .NET-a u Microsoftovim svijetu koristio ASP). Kažemo “zamjenu” jer Web Forms puno više sliči razvoju prozorskih aplikacija nego nadogradnji serverskog skriptiranja kakav smo koristili prije dolaska .NET-a.

Web-servisi predstavljaju web-aplikacije koje pružaju određenu funkcionalnost udaljenim programima. Primjerice, web-servis jedne banke može imati funkcionalnost preračunavanja iznosa iz eura u kune – vi njemu pošaljete iznos od 100 eura, a on vama vrati iznos od 770 kuna. Dakako, ovo je tek najjednostavniji primjer...

Radi jednostavnijeg snalaženja među klasama, one su raspoređene u hijerarhijsku strukturu koju možete vidjeti na slici uz tekst. Svaka stavka u strukturi ima svoje ime, koje nazivamo *name-*

Slika 2-3:
Shema biblioteka
klasa u .NET-u
zajedno s pripada-
jućim name-
spaceovima



space. Kao što u strukturi mapa na disku moramo znati točnu putanju da bismo došli do neke datoteke, tako i moramo znati točan *namespace* da bismo došli do određene klase. Evo primjera za dvije klase koje se jednako zovu, no nalaze se na drugim mjestima u hijerarhiji i, sukladno tome, imaju drugačije funkcije:

```
Using System.Web.UI.Control  
Using System.Windows.Forms.Control
```

Da bi nama programerima sve te mogućnosti bile nadohvat ruke i jednostavne za korištenje, postoji alat nazvan Visual Studio .NET. On sam po sebi ne pripada .NET Frameworku, no u potpunosti se na njega oslanja. U njegovu izvrsnom sučelju provodit ćete najviše vremena, osim ako za svoj razvojni alat ne izaberete neki drugi koji ima mogućnost rada u .NET okruženju, kao što je to Borlandov C# Builder.

Organizacija kôda

Uzmimo, primjera radi, da smo već napredni programeri i da smo napisali svoj prvi program. Nakon kompiliranja, rezultat koji smo dobili jest izvršna datoteka koja najčešće ima nastavak *.exe*, *.dll* ili *.netmodule*. Takva se izvršna datoteka zove upravljani modul (engl. *managed module*) i sastoji se od četiri osnovna elementa.

Da bi se spomenuta datoteka mogla izvršavati unutar Windows okruženja, njen prvi dio mora biti u formatu koji se zove Portable Executable (skraćeno PE). Osnovna uloga tog dijela je reći operativnom sustavu da se radi o upravljanom modulu te prebaciti kontrolu izvršavanja na CLR. Slijedi dio koji se naziva CLR zaglavlje, u kojem se zapisuju osnovni podaci o modulu. Treći dio čini MSIL-kôd, dok četvrti sadrži podatke o podacima, tzv. metapodatke.

Metapodaci

Da bi se određeni modul mogao koristiti, moramo znati što se u njemu nalazi i kako je sadržaj unutar njega organiziran. Tu u priču ulaze metapodaci koji opisuju modul i njegov sadržaj te na taj način uvelike olakšavaju korištenje modula.

U vremenu prije .NET-a nije postojala tako strogo definirana potreba za podacima koji opisuju podatke. Iako je u određenim slučajevima postojala ta mogućnost, ona nikad nije bila toliko opširna i, što je još važnije, nužna. Naime, metapodaci nisu opcionalna stvar – svaki upravljani modul mora sadržavati metapodatke, i na taj način svima zainteresiranim pružiti detaljnu informaciju o sebi.

I za interoperabilnost među programskim jezicima, koju smo već spominjali u kontekstu MSIL-a, vrlo je važno postojanje metapodataka. Oni, zahvaljujući standardiziranom načinu opisivanja modula,

I. DIO: .NET IZNUTRA

omogućavaju da moduli pisani u jednom jeziku koriste funkcionalnosti drugog, pisanog u tom ili nekom drugom jeziku.

Metapodaci se, osim za komunikaciju između modula, koriste i u gotovo svakom kutku .NET infrastrukture. Dio sistema koji je vjerojatno najviše zainteresiran za metapodatke je sam CLR. On pomoću njih radi provjeru valjanosti modula, aplicira sigurnosne parametre, planira korištenja memorije, povezuje s ostalim klasama koje modul koristi i na kraju pokreće izvršavanje. Bez metapodataka cijeli bi proces učitavanja i izvršavanja modula bio u najmanju ruku značajno složeniji i sporiji, ako ne i nemoguć.

Direktne koristi od metapodataka imaju i programeri. Primjerice, alati poput Visual Studia .NET koriste ih kako bi omogućili funkcionalnost automatskog kompletiranja riječi prilikom tipkanja koda, čime se smanjuje potreba za korištenjem dokumentacijom i pamćenjem brojnih naredbi.

Assemblies

Ukoliko naš upravljani modul ima određenu funkcionalnost koja će sama imati neku logičnu funkciju, onda ga istovremeno možemo zvati i *assembly*. Većina se *assemblyja* sastoji od samo jednog upravljanog modula, no postoje i oni koji u sebi ne samo da sadrže više upravljanih modula već i druge datoteke koje nisu moduli.

Najčešći slučaj u kojem nastaju *assemblyji* s više datoteka jest kada dva ili više modula pisanih u različitim jezicima povezujemo u jednu funkcionalnu cjelinu. Ta funkcionalna cjelina često ima potrebu i za nekim drugim resursima, poput slikovnih datoteka, koji također postaju dio *assemblyja*.

Prije nego što se izgubite u apstrakciji pojma “funkcionalna cjelina”, objasnimo što konkretno obuhvaća pojam *assemblyja*. Najlakše objašnjiva kategorija jest aplikacija. Dakle, u .NET terminologiji, aplikacija poput Notepada ili Microsoft Worda (pod uvjetom da je pisana u .NET-u) smatrala bi se *assemblyjem*.

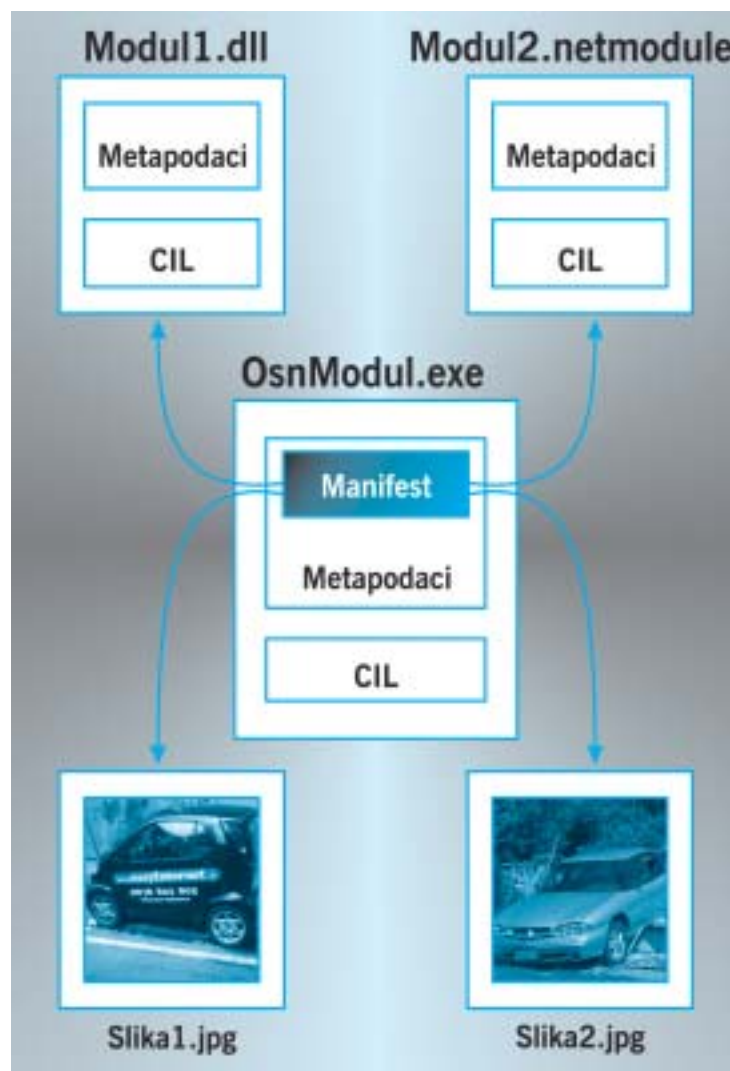
Nešto što smo dosada nazivali komponentama također pripada pod pojam *assemblyja*. Za razliku od prije spomenutih aplikacija, komponente najčešće nemaju korisničko sučelje i krajnji ih korisnici ne primjećuju, no zato aktivno sudjeluju u radu aplikacije koja ih koristi.

Primjer takvog odnosa može biti komponenta za provjeru pravopisa u aplikaciji za pisanje teksta. Iako se krajnjem korisniku to ne čini, radi se o dvije funkcionalne cjeline koje su međusobno odvojene i samostalne. Sve funkcionalnosti aplikacije (osim, dakako, provjere pravopisa) radit će i ako komponenta ne postoji, a funkcionalnosti komponente moći će koristiti i neka druga aplikacija, neka druga komponenta ili, sažeto sročeno, neki drugi *assembly*.

Manifesti

Da bi se znalo koje datoteke pripadaju u neki *assembly*, jedna od njih mora imati podatke o njemu i njegovim dijelovima. Ti se podaci nazivaju manifest i zapravo predstavljaju metapodatke o *assemblyjima*.

2. POGLAVLJE: ARHITEKTURA .NET-A



Slika 2-4:
*Shema assemblyja
 sastavljenog od tri
 upravljana modula i
 dvije slikovne datoteke*

Među tim podacima možemo pronaći ime *assemblyja* i listu datoteka koje čine *assembly*. Sadržaj svake datoteke je transformiran u šifrirani kontrolni izraz pomoću kojeg se može otkriti je li sadržaj datoteke mijenjan. Manifest sadrži i kopiju dijela metapodataka iz modula koje sadrži zajedno s informacijom na koji se modul odnose. Na taj način preko manifesta možemo direktno pristupiti funkcionalnostima pojedinih modula, i ne znajući da postoje.

Jedan od važnijih podataka koji se sprema u manifest jest njegova verzija. Ona se označava u formatu *major_version.minor_version.build.revision*, što u konačnici izgleda otprilike ovako: 1.2.2012.0. Osim broja verzije, tu može biti i takozvani *culture string* koji predstavlja zemlju i/ili jezik kojoj

I. DIO: .NET IZNUTRA

je *assembly* namijenjen (kulturni string za Hrvatsku je “hr-HR”), a osim njega možemo još pronaći podatke poput imena autora, opisa, potrebnih sigurnosnih prava i slično.



Želite li pregledati koje metapodatke i/ili manifeste sadrži neka datoteka, poslužite se pomoćnim programima koji dolaze uz .NET Framework SDK. Više o tim alatima potražite u dodacima na kraju knjige...

Postojanje manifesta u *assembly* donosi još jednu izvanrednu “nuspojavu”. Zahvaljujući činjenici da se u njemu nalaze svi podaci o *assembly*, nema potrebe za njihovom registracijom u sustav, kao što je to nužno s COM komponentama. Dovoljno ih je smjestiti na pravo mjesto na disku i mogu se koristiti.

Slabo i jako imenovanje

Imenovanje *assemblyja* posebna je priča. Imenovanje može biti slabo (engl. *weakly named*) i jako (engl. *strongly named*). Slabo imenovani *assembly* je onaj koji nije digitalno potpisan, a na njega se referencira koristeći samo ime navedeno u manifestu, što nije ništa drugo nego ime datoteke u kojoj se manifest nalazi napisano bez ekstenzije.

Jako imenovani *assemblyji* digitalno su potpisani, a na njih se referencira pomoću kombinacije imena, javnog ključa, oznake verzije i *culture stringa* ukoliko isti postoji. Bilo kakva promjena u tim parametrima motivirat će CLR da tretira *assembly* kao sasvim nov, ma koliko male stvarne promjene bile.

Uzmimo za primjer aplikaciju koja koristi slabi *assembly* u kojem ste primijetili neki propust. Otvorite ga u razvojnom alatu, popravljate uočenu pogrešku, kompilirate ga i dobivenu izvršnu datoteku snimate preko stare, i stvar radi dalje bez imalo problema.

Ukoliko se u istoj situaciji nađe jaki *assembly*, njegova će zamjena biti nešto kompliciranija. Naime, ako stavite novu inačicu, aplikacija će i dalje koristiti staru, a u slučaju da potonju izbrišete, aplikacija će otkazati poslušnost jer neće moći pronaći verziju *assemblyja* koju koristi.

S druge strane, situacija starijim programerima poznata pod nazivom “DLL Hell”, zahvaljujući jakom imenovanju, više ne postoji. Događa se, naime, da instalacija nekog novog softverskog paketa zamijeni neku od dijeljenih DLL-komponentata novom inačicom iste. U dobrom dijelu slučajeva ta zamjena prođe bez posljedica, no zna se dogoditi da neka prije instalirana aplikacija koja koristi tu komponentu otkáže poslušnost. To se događa u slučaju da nova verzija komponente nema sve mogućnosti koje su postojale u staroj ili su one tako nadograđene da ih se ne može koristiti

2. POGLAVLJE: ARHITEKTURA .NET-A

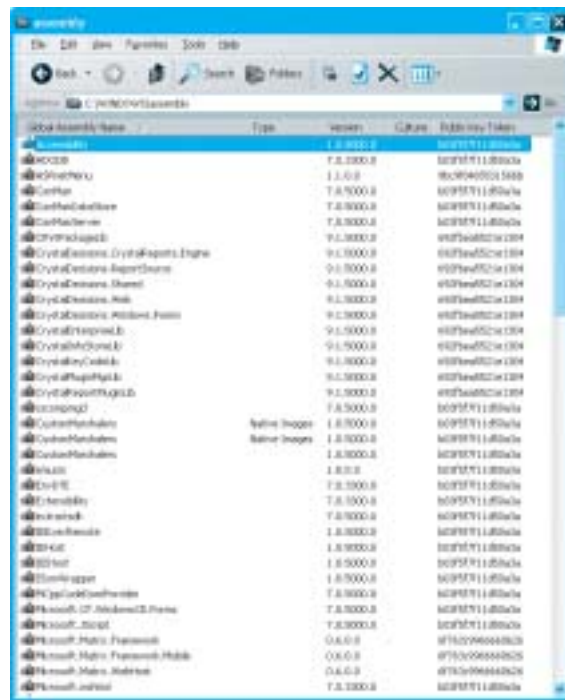
na isti način. Kako više nema funkcionalnosti koje stara aplikacija koristi, ona javlja grešku i otkaže poslušnost. Pokušate li novosnimljenu inačicu komponente zamijeniti starom, nova aplikacija neće raditi. Bezizlazna situacija u kojoj se tada nalazite zove se pakao DLL-a.

U .NET okruženju nema takvih problema. Zahvaljujući jakom imenovanju *assemblyja*, sistem dozvoljava koegzistenciju više različitih verzija iste komponente – svaka će aplikacija nastaviti koristiti svoju verziju, a vi nećete vidjeti niti plamičak vječne vatre...

Kao što vidite, oba tipa imenovanja *assemblyja* imaju svoje dobre i loše strane. Kako je na programeru da izabere koje će imenovanje koristiti, stvar ćete moći prilagoditi okolnostima. S druge strane, vrlo je vjerojatno da ćete u početku razvijati isključivo slabe *assemblyje*, a kasnije, kada se dohvatite jakih, verzioniranje vam više neće biti problem.

Global Assembly Cache

Tip imena za svojeg ljubimca ne možete birati kada ga želite podijeliti s ostatkom svijeta. Drugim riječima, želite li napraviti *assembly* koji će moći koristiti sve .NET aplikacije na računalu, morat ćete ga digitalno potpisati i staviti na mjesto koje se zove globalni spremnik *assemblyja* (engl. *Global Assembly Cache*, skraćeno GAC). GAC je svojevrsno spremište svih javno dostupnih *assemblyja* (pod "javno" se misli na lokalno računalo).



Slika 2-5:
Pogled na Global Assembly Cache i neke assemblyje koji se u njemu nalaze

I. DIO: .NET IZNUTRA

Razlog zbog kojeg GAC ne tolerira slabo imenovane *assemblyje* jest realna opasnost da se na istom računalu susretnu dva *assemblyja* istog (slabog) imena, a različitih proizvođača i funkcionalnosti.

Sada kada razumijemo pojam *assemblyja*, možemo reći da su sve one klase spomenute u priči o Base Class Libraryju i ostalim klasama u .NET Frameworku raspoređene po *assemblyjima* i da se nalaze u GAC-u. Zahvaljujući činjenici da su svi ti “ugrađeni” *assemblyji* jako imenovani, na istom se računalu može istovremeno instalirati i koristiti više inačica .NET Frameworka.

Garbage collection

I ako se radi o prilično naprednom sustavu koji po logici stvari ne bi trebao dobiti prostor u ovom letimičnom pregledu arhitekture .NET-a, ipak ćemo ga spomenuti. On, naime, može poslužiti kao izvrstan primjer prednosti upravljanog izvršavanja kôda unutar okruženja kao što je CLR.

Jedna od najnapornijih i dosadnijih zadata programera je traženje tzv. *memory leaks*, mjesta gdje određeni objekti zauzimaju memoriju iako se više ne koriste i bez posljedica bi mogli iz nje biti uklonjeni.

Takvo traženje igle u plastu sijena u .NET okruženju postaje stvar prošlosti. Za to se brine sustav zvan Garbage collection koji pregledava memoriju kad se napuni i iz nje briše stvari koje se više ne koriste. Automatski, i bez zahtjeva programera, dakako.

Proces počinje time što se sav sadržaj memorije proglašava nepotrebnim. Zatim kreće šetnja kroz aplikaciju i stvaranje liste svih dijelova memorije koji se koriste. Kad je lista gotova, preostali, nepopisani dio memorije smatra se nepotrebnim i biva obrisao. Djelici memorije koji su preživjeli čišćenje skupljaju se u hrpu na početku memorije, vrlo slično načinu na koji Windowsi rade defragmentaciju tvrdog diska.

Kao što smo već spomenuli, ovo je tek jedna od beneficija izvršavanja koda unutar upravljanog okruženja. CLR se, osim skupljanja smeća po memoriji, brine još za pregršt drugih manjih ili većih stvari, sve s ciljem da programiranje učini jednostavnijim i bržim.