2

Johdanto Microsoft .NETiin

Ilman vankkaa .NETin tuntemista ja tietoa siitä, miten C# toimii tässä Microsoftin rakenteessa, et voi täysin ymmärtää joitakin C#:n peruselementtejä, joita .NETin ajonaikainen ympäristö tukee. Tässä luvussa esitettävä .NETin yleiskatsaus auttaa sinua ymmärtämään tässä kirjassa käytettävän terminologian lisäksi sen, miksi määrätyt C#-kielen ominaisuudet toimivat niin kuin toimivat.

Jos olet lukenut .NETiä käsitteleviä uutisryhmiä tai postituslistoja, olet huomannut, että jotkut käyttäjät ovat turhautuneet uuden tekniikan terminologiaan. On vaikea pysyä ajan tasalla, koska epäselviä ja joskus ristiriitaisia sanoja heitellään esiin. Osan ongelmista aiheuttaa se, että tekniikka on kokonaan uutta. Ihan ensimmäiseksi määrittelen muutaman .NETin perustermin.

Microsoft .NET Platform

Microsoft .NETin perusajatus on siirtää painopiste tietojenkäsittelyyn maailmaan, jossa yksittäiset laitteet ja Web-sivustot on yhdistettu Internetin kautta toisiinsa, jolloin laitteet, palvelut ja tietokoneet toimivat yhdessä tuottaen monipuolisempia sovelluksia käyttäjille. Microsoft .NET koostuu neljästä perusosasta:

- .NET Building Block Services -palvelut eli ohjelmallinen pääsy määrättyihin palveluihin, kuten tietovarastoihin, kalenteriin ja Passport.NETiin (identiteetin tunnistuspalveluun).
- .NET laiteohjelmat, joita ajetaan Internet-laitteissa.
- .NET käyttäjäkokemus (user experience), joka sisältää mm. luonnollisen käyttöliittymän, informaatioagentit ja smart tagit, jotka muodostavat automaattisesti hyperlinkin käyttäjän dokumentissa oleviin sanoihin.

- .NET-rakenteen, joka sisältää seuraavat osat: .NET Framework, Microsoft Visual Studio.NET, .NET Enterprise Servers ja Microsoft Windows.NET.

.NET-rakenne on se osa .NETiä, johon useimmat ohjelmoijat viittaavat puhuessaan .NETistä.Voit olettaa, että aina kun puhun .NETistä (ilman edeltävää adjektiivia), tarkoitan nimenomaan .NET-rakennetta. .NET-rakenne viittaa kaikkiin niihin tekniikoihin, joista muodostuu uusi ympäristö luotettaville, skaalautuville ja hajautetuilla sovelluksille. Se osa .NETiä, jonka avulla voimme näitä sovelluksia tehdä, on .NET Framework.

.NET Framework koostuu CLR:stä (Common Language Runtime) ja .NET Framework -luokkakirjastoista, jotka tunnetaan myös nimellä Base Class Library (BCL). Voit ajatella CLR:ää virtuaalikoneena, jossa .NET-sovellukset toimivat. .NET Framework luokkakirjastot ovat kaikkien .NET-kielten käytössä. Jos tunnet joko Microsoft Foundation Classes (MFC) tai Borlandin Object Windows Library (OWL) -kirjastot, tunnet luokkakirjastot. .NET Framework luokkakirjastoihin sisältyy tuki kaikelle tiedostokäsittelystä ja tietokantakäsittelystä XML:ään ja SOAPiin. Itse asiassa .NET Framework luokkakirjastot ovat niin laajoja, että vaatisi kokonaisen kirjan, jotta sen kaikista luokista pystyisi antamaan edes pinnallisen kuvauksen.

Sanon sivuhuomautuksena (yhtä hyvin kuin oman ikänä tunnustuksena), että kun käytän termiä "virtuaalikone", en tarkoita Java Virtual Machine (JVM):tä. Käytän vain termin alkuperäistä merkitystä. Muutamia vuosikymmeniä sitten, kun Java oli vain musta ja kuuma juoma, IBM puhui virtuaalikoneesta. Se oli korkean tason käyttöjärjestelmän abstraktio, jonka sisällä muut käyttöjärjestelmät pystyivät toimimaan täysin eristetysti. Kun viittaan CLR:ään virtuaalikoneena, tarkoitan sitä, että CLR:ssä suoritettava koodi ajetaan eristetyssä ja hallitussa ympäristössä, erillään koneen muista prosesseista.

.NET Framework

Katsotaan mikä on .NET Framework ja mitä se tarjoaa. Ensimmäiseksi vertaan sitä muihin hajautettuihin sovelluskehitysympäristöihin. Seuraavaksi käyn läpi ne ominaisuudet, jotka .NET Framework tarjoaa sovelluskehittäjille tehokkaiden hajautettujen sovellusten entistä nopeampaan kehittämiseen.

Windows DNA ja .NET

Kuulostiko fraasi, jolla hetki sitten kuvailin .NETiä eli "uusi ympäristö luotettavien, skaalautuvien ja hajautettujen sovellusten tekemiseen ja suorittamiseen", tutulta? Jos tuntui, syy on seuraava: .NET on pohjimmiltaan jälkeläinen aiemmille yrityksille tyydyttää nämä

ylevät periaatteet. Sitä edellistä alustaa kutsutaan nimellä Windows DNA. Se on ratkaisualusta, joka keskittyi ratkaisemaan liiketoimintaongelmat Microsoftin palvelintuotteiden avulla. Termiä "liima" on joskus käytetty Windows DNA:n yhteydessä, esimerkiksi "DNA määrittelee liiman, jota käytetään kokoamaan yhteen luotettava, skaalautuva ja hajautettu järjestelmä." Kuitenkin, lukuunottamatta teknisiä määrittelyjä, Windows DNA:lla ei ollut konkreettisia osia. Tämä on yksi mutamasta suuresta Windows DNA:n ja .NETin välisestä erosta. Microsoft .NET ei ole pelkästään joukko määrityksiä. Siihen sisältyy myös konkreettisia tuotteita, kuten kääntäjiä, luokkakirjastoja ja jopa kokonaisia loppukäyttäjän sovelluksia.

Common Language Runtime

CLR on .NETin ydin. Kuten nimikin viittaa, se on ajonaikainen ympäristö, jossa eri kielillä tehdyt sovellukset voivat kaikki toimia hienosti yhdessä (cross-language interoperability). Miten CLR tarjoaa tämän mukavan ymäristön eri kielten väliselle yhteistoiminnalle? Common Language Specification (CLS) on joukko sääntöjä, jotka ohjelmointikielen kääntäjän tulee täyttää, jotta se voisi tehdä CLR:ssä toimivia .NET-sovelluksia. Jokainen, jopa sinä tai minä, joka haluaa kirjoittaa .NET-yhteensopivan kääntäjän, tulee noudattaa näitä sääntöjä, ja katso, kääntäjäsi generoimat sovellukset toimivat oikein minkä tahansa muun .NET-sovelluksen kanssa ja hyödyntävät samaa yhteistoimintaominaisuutta.

Tärkeä CLR:ään liittyvä käsite on *hallittu koodi* (managed code). Se tarkoittaa koodia, joka suoritetaan CLR:n alaisuudessa ja josta siten voi sanoa, että CLR hallitsee sitä. Ajattele sitä näin: Nykypäivän Microsoft Windows -ympäristössä meillä on käynnissä erilaisia prosesseja. Ainoa sääntö, jota sovellusten tulee noudattaa, on se, että ne käyttäytyvät hyvin Windows-ympäristössä. Nämä sovellukset luodaan monilukuisten ja täysin erilaisten kääntäjien avulla. Toisin sanoen, sovellusten tulee noudattaa vain yleisintä sääntöä toimiakseen Windowsissa.

Windows-ympäristössä on vain muutamia yleisiä sääntöjä, jotka määräävät miten sovellusten tulee käyttäytyä "keskustellessaan" muiden kanssa, varatessaan muistia tai jopa pyytäessään Windows-käyttöjärjestelmää tekemään jotain puolestaan. Hallitun koodin ympäristössä sen sijaan on joukko sääntöjä varmistamaan, että kaikki sovellukset käyttäytyvät yleisellä ja yhtäläisellä tavalla riippumatta kielestä, jolla ne on kirjoitettu. Yhtäläinen sovellusten käyttäytyminen on .NETin olennainen tekijä eikä sitä voi liioitella. Onneksi meidän kannaltamme, nämä yleiset säännöt koskettavat vain kääntäjien tekjöitä.

.NET Framework -luokkakirjastot

.NET Framework -luokkakirjastot ovat äärimmäisen tärkeitä kielten yhteistoiminnan kannalta, koska niiden avulla ohjelmoijat käyttävät yhtä ohjelmointirajapintaa kaikkiin CLR:n tarjoamiin palveluihin. Jos olet joskus käyttänyt useampaa kuin yhtä kieltä Windows-ohjelmointiin, rakastat tätä ominaisuutta. Itse asiassa .NET Framework luokkakirjastot muovaavat uuden vallankumouksellisen suuntauksen kääntäjien teossa. Ennen .NETiä useimmat kääntäjien ohjelmoijat tekivät kielen, jolla oli kyky tehdä suurin osa omasta työstään. Jopa C++:n tapaisilla kielillä, jotka on suunniteltu käytettäväksi yhdessä luokkakirjaston kanssa, on itsessään vähintään joitain perustoimintoja. .NET-maailmassa kielistä tulee syntaksisia rajapintoja .NET Framework luokkakirjastoihin.

Katsotaan esimerkkinä perinteistä "Hello, world"-sovellusta C++-kielellä ja verrataan sitä sovellukseen, joka tekee saman C#:lla:

```
#include <iostream.h>

int main(int argc, char* argv[])
{
    cout << “Hello, World!” << endl;
    return 0;
}
```

Huomaa, että sovellus ottaa ensin mukaan otsikkotiedoston, jossa on *cout*-funktion määrittely. Sovelluksen *main*-funktio, joka on jokaisen C/C++-sovelluksen alkukohta, käyttää *cout*-funktiota kirjoittaakseen merkkijonon “Hello, World” vakiotulostuslaitteelle. Tärkeää huomata tässä on kuitenkin se, että et voi kirjoittaa tätä sovellusta millään .NET-kielellä ilman .NET Framework luokkakirjastoja. Aivan niin, .NET kielillä ei ole edes kääntäjien perusominaisuuksia, kuten mahdollisuutta kirjoittaa merkkijonoa ikkunaan. Tiedän, että teknisesti *cout*-funktio on toteutettu C/C++:n ajonaikaisessa ympäristössä, joka itsessään on kirjasto. Kuitenkin C++:n perustehtävät, kuten merkkijonon muotoilu, tiedosto I/O ja näyttö I/O ovat, ainakin loogisesti, ajateltava olevan osa peruskieltä. C#:lla, tai millä tahansa .NET-kielellä, ei itsellään ole minkäänlaista kykyä tehdä yksinkertaisimpiakaan asioita ilman .NET Framework luokkakirjastoa.

Katsotaan nyt “Hello, World”-esimerkkiä C#:lla, niin näet mitä tarkoitan:

```
using System;

class Hello
{
    public static void Main()
    {
        Console.WriteLine(“Hello, World”);
    }
}
```

Eli, mitä tämä luokkakirjastojen joukko merkitsee ja onko se hyvä asia? No, se riippuu näkökulmastasi. Yleiset luokkakirjastot merkitsevät, että teoreettisesti kaikilla kielillä on samat ominaisuudet, koska niiden kaikkien pitää käyttää näitä luokkakirjastoja tehdessään jotain, lukuunottamatta muuttujien määrittelyä.

Yksi mielipide, jonka olen nähnyt keskusteluryhmissä on: "Miksi kieliä on useita, jos niillä kaikilla on samat ominaisuudet?" En ymmärrä tätä valitusta. Kuten kaikki, jotka ovat työskennelleet monikielisessä ympäristössä, voin minäkin todistaa, että on suuri etu, kun ei tarvitse muistaa millä kielellä voi tehdä mitäkin ja miten se sen tekee. Loppujen lopuksi tehtävämme ohjelmoijina on tuottaa koodia, ei miettiä, onko suosikkikielessä sitä tai tätä ominaisuutta.

Toinen usein tehty kysymys kuuluu: "Jos nämä kaikki .NET-kielet pystyvät samaan, miksi tarvitsemme niitä enemmän kuin yhden?" Vastaus perustuu siihen tosiasiaan, että ohjelmoija on tapojensa orja. Microsoft ei missään tapauksessa halunnut valita yhtä kieltä ja pakottaa miljoonat ohjelmoijat unohtamaan vuosien kokemuksensa muista kielistä. Ohjelmoijan ei pitäisi ainoastaan tutustua uuteen APIiin, vaan myös opetella täysin erilainen kielen syntaksi. Nyt ohjelmoija voi jatkaa tehtävään parhaiten sopivan kielen käyttämistä. Loppujen lopuksi oleellista on tuottavuus. Tarpeettoman muuttaminen ei kuulu siihen.

Huomaa Vaikka teoriassa .NET Framework luokkakirjastot sallivat kaiken CLR:n toiminnallisuuden toteuttamisen eri kielten käyttäjälle, ei asia aina ole näin. Yksi riidan aihe Microsoftilla .NET Framework luokkakirjaston tekijöiden ja eri kielen kääntäjien tekijöiden kesken on se, että vaikka .NET Framework luokkakirjaston tekijät ovat yrittäneet laajentaa kaiken toiminnallisuuden kaikkiin kieliin, ei ole mitään, lukuunottamatta CLS-standardien täyttäminen, joka vaatii, että eri kääntäjien tulee toteuttaa jokainen yksittäinenkin ominaisuus. Kun kysyin tästä erilaisuudesta muutamalta Microsoftin ohjelmoijalta, minulle kerrottiin, että sen sijaan, että jokaisen kääntäjän tulisi saavuttaa jokainen .NET Frameworkin toiminto, kukin kehitystiimi on päättänyt toteuttaa vain ne ominaisuudet, jotka heidän mielestään ovat sopivimpia heidän käyttäjilleen. Meidän onneksemme C# tuntuu olevan kieli, joka tarjoaa rajapinnan lähes kaikkiin .NET Frameworkin toimintoihin.

Microsoft Intermediate Language ja JITterit

Tehdäkseen kielten siirtämisen .NETiin helpoksi, Microsoft kehitti assembly-kielen sukulaisen nimeltä Microsoft intermediate language (MSIL). Kääntäessään sovelluksia .NETiin, kääntäjät lukevat lähdekoodin ja kääntävät sen MSIL:ksi. MSIL itse on täydellinen kieli, jolla voit kirjoittaa sovelluksia. Mutta kuten assembler-kielenkin kanssa, et luultavasti koskaan kirjoita sillä mitään. Koska MSIL on oma kielensä, kunkin kääntäjän kehitystiimi tekee oman päätöksensä siitä, miten paljon se tukee MSIL:ää. Jos kuitenkin olet kääntäjän tekijä ja haluat tehdä kielen, joka toimii yhdessä muiden kielten kanssa, sinun tulee rajoittaa itsesi CLS:ssä määriteltyihin määrityksiin.

Kun käännät C#-sovelluksen, tai millä tahansa CLS-sopivalla kielellä tehdyn sovelluksen, se käännetään MSIL:ksi. Tämä MSIL käännetään edelleen aidoksi konekieleksi, kun CLS käynnistää sovelluksen ensimmäisen kerran. (Itse asiassa vain kutsuttavat funktiot käännetään, kun niitä kutsutaan ensimmäisen kerran.) Koska meissä jokaisessa kuitenkin asuu pieni nörtti, katsotaan, mitä konepellin alla todella tapahtuu:

1. Kirjoitat lähdekoodin C#:lla.
2. Käännät sen .EXE-tiedostoksi C#-kääntäjällä (csc.exe).
3. C#-kääntäjä kirjoittaa MSIL-koodin ja luettelon (manifest) EXE:n vain-lukuosaan, jolla on standardin mukainen PE:n (Win32-portable executable) otsikko-osa.

   Tähän asti kaikki hyvin. Tässä on tärkein osa: kun kääntäjä tekee tiedoston, se ottaa mukaan .NETin ajonaikaisesta kirjastosta funktion nimeltä *_CorExeMain*.
4. Kun sovellus käynnistetään, käyttöjärjestelmä lataa PE:n sekä kaikki tarvittavat DLL:t (dynamic-link library), kuten sen, joka toteuttaa *_CorExeMain*-funktion (mscoree.dll), aivan kuten se tekee jokaisen kelvollisen PE:n kanssa.
5. Käyttöjärjestelmän lataaja hyppää PE:n sisällä aloituskohtaan, jonka sinne on sijoittanut C#-kääntäjä. Tämä on edelleen aivan sama toiminto kuin minkä tahansa PE:n suorituksessa Windowsissa.

   Koska käyttöjärjestelmä ei tietenkään pysty suorittamaan MSIL-koodia, aloituskohta onkin hyppy mscoree.dll:n *_CorExeMain*-funktioon.
6. *_CorExeMain*-funktio käynnistää PE:hen sijoitetun MSIL-koodin suoritukseen.

7. Koska MSIL-koodia ei voida suorittaa suoraan, sillä sehän ei ole konekielisenä, kääntää CLR MSIL:n konekieleksi käyttämällä täsmäkääntäjää (just-in-time compiler, JIT) eli JITteriä. Täsmäkäännös tehdään vain, kun ohjelman metodeja kutsutaan. Käännetty konekielinen koodi tallennetaan koneelle ja se käännetään uudelleen vain, jos lähdekoodiin on tehty muutoksia.

MSIL:n muuttamisessa konekieleksi voidaan käyttää tilanteen mukaan kolmea eri JITteriä:

- **Asennusaikainen koodin generointi, PreJIT** Asennusaikainen koodin generointi kääntää koko koosteen prosessorikohtaiseksi binäärikoodiksi, aivan kuten C++-kääntäjä tekee. Kooste (assembly) on koodipaketti, joka annetaan kääntäjälle. (Puhun koosteista tarkemmin tämän luvun kappaleessa "Jakelu"). Tämä käännös tehdään asennuksen yhteydessä, jolloin loppukäyttäjä tuskin edes huomaa, että kooste käännetään samalla. Asennusaikaisen koodin generoinnin etu on siinä, että käännös tehdään vain kerran ennen ensimmäistä käyttökertaa. Koska koko kooste käännetään, sinun ei tarvitse huolehtia ajoittaisista suorituskyvyn heikennyksistä joka kerta, kun metodisi ensimmäisen kutsun yhteydessä käännetään. Se on kuin lomamatka, jonka maksat ennakkoon. Vaikka maksun suorittaminen on pieni painajainen, et loman aikana enää joudu huolehtimaan siitä. Kannattaako sinun käyttää tätä mahdollisuutta, riippuu järjestelmästäsi ja ohjelman jakeluympäristöstä. Normaalisti, jos olet tekemässä sovelluksen asennusta järjestelmääsi, sinun kannattaa käyttää tätä JITteriä, jotta käyttäjät saavat käyttöönsä ohjelmastasi täysin optimoidun version.
- **JIT** Tätä JITteriä kutsutaan suorituksen aikana edelläkuvatulla tavalla, joka kerta, kun metodia kutsutaan ensimmäisen kerran. Tämä vaihtoehto on eräänlainen "maksa-kun-käytät"-periaate. Tämä on myös oletustoiminto, jos et erikseen määrittele PreJIT-käännöstä.
- **EconoJIT** Toinen ajonaikainen JITter, EconoJITter, on erityisesti suunniteltu kevyitä järjestelmiä varten. Tällaisia ovat esimerkiksi kämmentietokoneet, joissa on vähän muistia. Suurin ero tämän ja tavallisen JITterin välillä on koodin hylkäys (code pitching). Sen ansiosta EconoJITter voi hylätä generoidun koodin, jos järjestelmän muisti alkaa olla vähissä. Haittana tästä on se, että jos hylätty koodi aktivoidaan uudelleen, se pitää taas kääntää kuin sitä ei olisi ennen kutsuttu.

Yleinen tyyppijärjestelmä

Yksi kehitysympäristön avainominaisuuksia on sen tyyppijärjestelmä. Loppujen lopuksi kehitysympäristö, jossa on rajoittunut määrä tyyppejä tai järjestelmä, joka rajoittaa ohjelmoijan mahdollisuutta laajentaa järjestelmän tarjoamia tyyppejä, ei tule elämään kauaa. .NET tarjoaa ohjelmoijalle enemmän kuin pelkän yhden yhdistetyn tyyppijärjestelmän, joka on kaikkien CLR-yhteensopivien kielten käytettävissä. Se sallii kielten ohjelmoijien laajentavan tyyppijärjestelmää lisäämällä uusia tyyppejä, jotka näyttävät ja toimivat samoin kuin järjestelmän omat tyypit. Tämä tarkoittaa, että sinä ohjelmoijana voit käyttää *kaikkia* tyyppejä yleisellä tavalla riippumatta siitä, ovatko ne .NETin valmiita tyyppejä tai ohjelmoijan luomia. Kerron tyyppijärjestelmän yksityiskohdista ja siitä, miten C#-kääntäjä tukee sitä, luvussa 4, "Tyyppijärjestelmä."

Metadata ja reflection-menetelmä

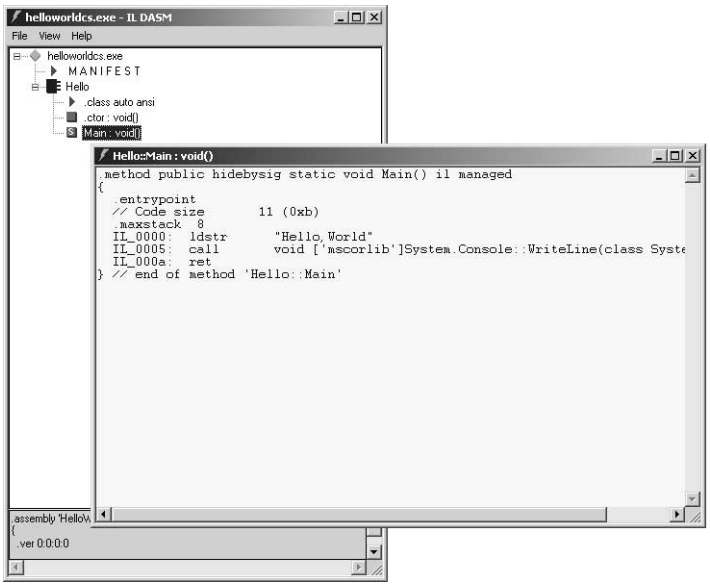
Kuten aiemmin mainitsin kappaleessa “Microsoft Intermediate Language ja JITterit,” CLS-yhteensopivat kääntäjät ottavat syötteekseen lähdekoodin ja tuottavat MSIL:n käännettväksi ajon aikana (JITterillä) ja suoritettavaksi. Sen lisäksi, että kääntää lähdekoodin MSIL:n käskyiksi, on kääntäjällä toinen yhtä tärkeä tehtävä: upottaa metadata lopputuloksena syntyvään EXE-tiedostoon.

Metadata on tiedot, jotka kuvaavat jotain. Tässä yhteydessä se on joukko EXE:n muodostavia asioita, kuten tyyppien määritykset ja metodien toteutukset. Jos tämä tuntuu tutulta, niin sen kuuluukin tuntua. Metadata on samanlainen asia kuin tyyppikirjastot, jotka generoituvat yhdessä Component Object Model (COM) -komponenttien kanssa. .NET kääntäjän muodostama metadata on monipuolisempi ja täydellisempi kuin COMin tyyppikirjastot ja se on lisäksi aina upotettu EXE:een. Tämän ansiosta se ei ikinä häviä eikä järjestelmässä ole epäyhteensopivia pareja.

Syy metadatan käyttämiseen on yksinkertainen. Sen ansiosta .NETin ajonaikainen ympäristö tietää ohjelman suorituksen aikana, mitä tyyppejä käytetään ja mitä metodeja kutsutaan. Tämän ansiosta ajonaikainen ympäristö voidaan muodostaa oikein, jolloin ohjelma suoritetaan tehokkaasti. Menetelmää, jolla metadata selvitetään, kutsutaan nimellä reflection. Itse asiassa .NET Framework luokkakirjasto sisältää koko joukon reflection-metodeja, joidan avulla CLR:n lisäksi jokainen sovellus voi kysellä sovelluksen metadataa.

Ohjelmointiympäristöt käyttävät näitä reflection-metodeja toteuttaessaan IntelliSensen tapaisia ominaisuuksia. IntelliSense toimii niin, että kun kirjoitat metodin nimen, sen parametriluettelo ilmestyy ruudulle. Visual Studio.NET vie tämän vieläkin pidemmälle näyttäen myös tyypin jäsenet. Puhun reflection-APIsta luvussa 16, "Metadatan kyseleminen reflection-metodien avulla."

Toinen äärimmäisen käyttökelpoinen .NET-työkalu, joka hyödyntää reflection-metodeja, on Microsoft .NET Framework IL Disassembler (ILDASM). Tämä tehokas apuohjelma jäsentää sovelluksen metadatan ja näyttää sen puumaisena rakenteena. Kuvassa 2-1 näet, miltä "Hello, World"-sovellus näyttää ILDASM-ikkunassa.



Kuva 2-1 C# "Hello, World" -sovellus ILDASM-ikkunassa.

Ikkuna kuvan 2-1 taustalla on IL Disassemblerin pääikkuna. *Main*-metodin kaksoisnapautus avaa etualalla olevan ikkunan, joka näyttää*Main*-metodin yksityiskohtaiset tiedot.

Turvallisuus

Tärkein näkökohta missä tahansa hajautettujen sovellusten kehitysympäristössä on turvallisuus. Jo kauan on sanottu, että Microsoftia ei tulla koskaan ottamaan vakavasti palvelinpuolen hajautettujen sovellusten alustana, jos se ei ota täysin uutta asennetta turvakysymyksiin. .NET iskee pöytää monia ratkaisuja. Turvallisuus alkaa siitä hetkestä, kun CLR lataa luokan, koska luokan lataaja on osa .NETin turvaverkkoa. Esimerkiksi, kun luokka on ladattu .NETin ajonaikaiseen ympäristöön, tarkistetaan sen turva-asetuksiin liittyvät ominaisuudet, kuten käsittelyoikeudet ja koostumus. Lisäksi turvatarkistus varmistaa, että tietyllä koodin osalla on valtuudet määrättyjen resurssien käsittelyyn.

Turvakoodi varmistaa roolien määrittelyn ja identiteettitiedot. Nämä tarkistukset jopa ylittävät prosessien ja laitteiden rajat varmistaakseen, että luottamukselliset tiedot eivät vaarannu hajautetussa ympäristössäkään.

Ohjelmien jakelu

Ohjelmien jakelu on tähän asti ollut hirvittävin tehtävä laajoissa hajautetuissa järjestelmissä. Itse asiassa, kuten jokainen Windows-ohjelmoija voi vahvistaa, tappelu erilaisten binaaritiedostojen, rekisteröintiongelmien, COM-komponenttien ja tarpeellisten tukikirjastojen asennuksien kanssa on saanut monen miettimään uravalintaansa uudelleen. Onneksi ohjelmien jakelu on seikka, johon .NETin kehitystiimi on erityisesti paneutunut.

Avain .NETin ohjelmien jakelussa on kooste (assembly). Se on yksinkertainen paketti toisiinsa liittyviä asioita, jotka muodostuvat joko yhdestä tai useasta tiedostosta. Määritykset, miten jakelet sovelluksesi, riippuu siitä, oletko tekemässä Web-palvelinsovellusta tai perinteistä pöytäkonesovellusta Windowsiin. Yksinkertaisimmillaan ohjelman jakelu on tarpeellisten koosteiden kopioiminen kohdekoneen kansioon.

Monet niistä ongelmista, jotka aiheuttivat paljon vaivaa ennen NETiä, on nyt eliminoitu. Esimerkiksi komponentin rekisteröintiä ei enää tarvita, kuten piti tehdä COM-komponenteille ja ActiveX-komponenteille. Syy on se, komponentit pystyvät kuvaamaan itsensä metadatan ja reflection-metodien avulla. .NETin ajonaikainen ympäristö myös pitää kirjaa sovellukseen liittyvistä tiedostoista ja niiden versioista. Siksi jokaiseen asennettuun sovellukseen liitetään automaattisesti sen koosteen muodostavat tiedostot. Jos sovelluksen asennus yrittää korvata toisen sovelluksen tarvitseman tiedoston, .NET on tarpeeksi fiksu salliakseen sovelluksen asentaa tarvittavat tiedostot, mutta CLR ei tuhoa tiedoston edellistä versiota, koska ensimmäinen sovellus tarvitsee sitä yhä.

Yhteistoiminta hallitsemattoman koodin kanssa

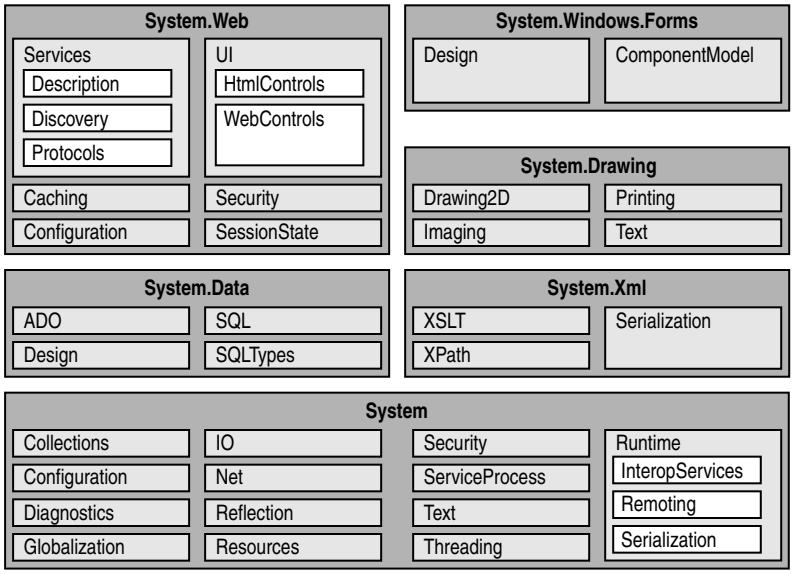
Kuten varmaan arvaat, *hallitsematon koodi* (unmanaged code) on sellainen koodi, jota .NETin ajonakainen ympäristö ei kontrolloi. Selvyyden vuoksi sanottakoon, että se kuitenkin suorittaa tällaisen koodin. Hallitsemattomalla koodilla ei kuitenkaan ole niitä ominaisuuksia, joita hallitulla koodilla on, kuten roskienkeruu, yleinen tyyppijärjestelmä ja metadata. Ihmettelet varmaan, miksi kukaan haluaisi suorittaa hallitsematonta koodia .NETissä.

Joskus ei kuitenkaan ole mahdollisuutta muuhun. Seuraavassa muutamia tilanteita, joissa olet kiitollinen Microsoftille, että se otti tällaisen ominaisuuden mukaan .NETiin:

- **Hallittu koodi kutsuu hallitsemattomia DLL-funktioita** Sanotaan, että sovelluksesi tarvitsee rajapinnan C-kielellä tehtyyn DLL:ään ja yritys, joka kirjoitti sen, ei ole siirtynyt .NETiin yhtä nopeasti kuin oma yrityksesi. Tällöin sinun tulee pystyä kutsumaan tuota DLL:ää .NET-sovelluksesta. Käsittelen tätä esimerkissä luvussa 16, "Metadatan kyseleminen reflection-metodien avulla."
- **Hallittu koodi käyttää COM-komponentteja** Samasta syystä, jonka vuoksi tarvitset mahdollisuuden kutsua C-kielisen DLL:n funktioita .NET-sovelluksesta, saatat tarvita tukea COM-komponenttien käytölle. Teet sen luomalle .NET-wrapperluokan COM-komponentille, jolloin hallittu koodisi kuvittelee työskentelevänsä .NET-luokan kanssa. Myös tästä puhutaan enemmän luvussa 16.
- **Hallitsematon koodi käyttää .NET-palveluja** Tämä on täsmälleen päinvastainen ongelma, haluat käyttää .NETiä hallitsemattomasta koodista. Se on ratkaistu vastavuoroisesti niin, että COM-asiakasohjelmalle luulotellaan, että se käyttää COM-palvelinkomponenttia, joka onkin tosiasiallisesti .NET-palvelu. Näet esimerkin myös tästä luvussa 16.

Yhteenveto

Microsoft .NET merkitsee siirtymistä tietojenkäsittelymalliin, jossa laitteet, palvelut ja tietokoneet toimivat yhdessä tarjoten ratkaisuja käyttäjille. Siirtymän olennainen osa on .NET Framework ja CLR, jonka näet seuraavan sivun kuvassa 2-2. .NET Framework sisältää luokkakirjastot, joita käyttävät ne kielet, joilla käännetään ohjelmia CLR:lle. Koska C# on suunniteltu CLR:ää varten, et voi suorittaa yksinkertaistakaan tehtävää ilman CLR:ää ja .NET Framework luokkakirjastoja. Näiden ymmärtäminen on tarpeen, jotta saat kaiken irti C#:sta ja kirjan loppuosasta.



Kuva 2-2 .NET Framework sisältää kirjastot, jotka on suunniteltu helpottamaan palvelujen, laitteiden ja tietokoneiden yhteistoimintaa.