MÄLARDALENS HÖGSKOLA
Institutionen för ekonomi och informatik

[A R B E T S K O P I A]

Komponenter med MTS/COM+ i .NET

med introduktion till Remoting och ASP.NET

EI0230 – Komponentbaserad applikationsutveckling

Björn Persson, november 2003

Om denna sammanfattning

Denna sammanfattning beskriver hur man använder de tjänster som Microsoft Transaction Server (MTS) och komponenttjänster (*Component Service*, COM+) ger. Observera att vissa beskrivningar kan vara ofullständiga, bl.a. för att spara plats eller för att de inte är relevanta för en grundläggande förståelse av MTS/COM+ och.NET, och att det **inte är en ersättning för eventuell kurslitteratur**. Denna sammanfattning utgår från sammanfattningen *Komponenter med DCOM/MTS*. För att få ut mest av denna sammanfattning så krävs det en grundläggande förståelse för Visual Basic 6/.NET, databaser/SQL, objektorienterad programmering och COM/MTS (eller COM+).

Denna sammanfattning bygger på VB.NET (och ibland C#), men mycket av stoffet bör även kunna appliceras i språk så som Managed C++ (det tredje språket som följer med Visual Studio.NET, VS.NET) eller andra .NET-språk.

Koden i denna sammanfattning har skrivits i Microsofts *Visual Studio.NET 1.0* (i Windows XP, d.v.s. COM+) och sedan kopierats in i dokumentet. Fel kan dock ha införts av misstag vid redigering av dokumentet. Övrig programvara från Microsoft som använts är *Access XP* (samt *Visio XP* för klassdiagram).

Konventioner i sammanfattning

I texten visas metoder med parenteser efter (t.ex. **Print()**) för att visa att det är en metod. Metoder och egenskaper som tillhör ett objekt visas med fet stil och punkt före (t.ex. **.Open()** resp. **.Source**) för att visa att dom är en del av objektet som stycket eller avsnittet behandlar.

Vissa engelska begrepp saknar (enligt mig) generellt accepterade översättningar och är därför skrivna på engelska för att kunna relatera till begreppen i engelsk litteratur. I de fall då översättning används så följs det översatta ordet första gången med det engelska inom parentes. Dessa ord skrivs i kursiv stil för att visa att de har ”lånats”, t.ex. *data provider* och datakällor (*data sources*). Kursiv stil används även för hänvisningar till kapitel, avsnitt eller andra sammanfattningar.

Kod har skrivits med typsnitt av fast bredd (`Courier New`) för att göras mer lättläst samt längre exempel har inneslutits i en ram (se exempel nedan).
I Visual Basic kan programsatser (*statements*) skrivas på flera rader genom att använda understrykningstecknet (”_”). Detta underlättar läsning av kod då man slipper skrolla i sidled för att läsa längre programsatser. Viktigt är att placera ett mellanslag innan understrykningstecknet som i detta exempel:

```
enVariabel = 1 _
             + 2    'Kommentarer i kod skriv med fet stil
```

I kapitel om distribuerade komponenter används ordet ”komponenttjänster” för att syfta till de tjänster som MTS/COM+ ger och ordet ”Komponenttjänster” (med versal i början) för att syfta på gränssnittet som används för att administrera COM+-applikationer i Windows 2000/XP.

Jag är givetvis tacksam för alla konstruktiva synpunkter på sammanfattningens utformning och innehåll.

Eskilstuna, november 2003

Björn Persson, e-post: `bjorn.persson@mdh.se`
Mälardalens högskola
Institutionen för ekonomi och informatik
Personlig hemsida: `http://www.eki.mdh.se/personal/bpn01/`

Innehållsförteckning

(Denna sida har avsiktligt lämnats tom.)

ATT GÖRA:

* Mycket…

1 .NET Framework

.NET Framework är en infrastruktur (plattform) för exekvering av applikationer, d.v.s. måste vara installerad på dator för både utvecklare och användare av program. Vi kan installera .NET Framework separat eller då vi installerar VS.NET. [1] I.o.m. service pack 1 till Windows XP så ges även möjligheten att installera .NET Framework samtidigt. I skrivande stund (2003-10-16) så är version 1.1 den senaste versionen av .NET Framework.

För att utveckla program kan vi använda enbart **.NET SDK**[2] (som går att ladda ner från Microsofts webbplats) eller VS.NET (som måste köpas ☹). I det första fallet kan vi använda vilken texteditor som helst och sen kompilera koden med de kommandobaserade kompilatorerna. Använder vi VS.NET så får en massa andra verktyg på kuppen. Det finns också ett antal gratis miljöer (IDE:er) så som #develop (uttalas *sharp develop*) – en IDE som främst är riktad till C#-programmerare, men som till viss mån även fungerar med t.ex. VB.NET och andra .NET-språk.

Kompilatorerna genererar inte maskinkod direkt, utan kod av typen *Intermediate Language* (IL). IL-koden distribueras dock i EXE- eller DLL-filer, vilka kallas *.NET Portable Executables* (PE). När sen program exekverar och anropar metoder i klasser så kompilerar JIT-kompilatorn (*Just-In-Time Compiler*) metodernas IL-kod till maskinkod. Maskinkoden buffras tills den inte behövs längre (t.ex. process avslutas). IL-kod för metoder som inte anropas kompileras alltså inte.

PE-filerna innehåller metadata som beskriver de datatyper som finns i filen samt vilka referenser till andra PE-filer som finns. Denna metadata används av .NET Framework för att bl.a. avgöra vilka PE-filer som behöver laddas.

I .NET Framework ingår Common Runtime Library (CLR), Common Base Library (CBL) och Common Type System (CTS).

1.1 Common Runtime Library (CLR)

CLR är programvaran som används för att exekvera .NET-program. I CLR finner vi bl.a. klassladdaren, verifieraren, JIT-kompilatorn samt annan mjukvara för att stödja exekveringen av .NET-program (bl.a. *garbage collection*). Observera att CLR **inte** är en virtuell maskin (så som Javas JVM), även om det finns stora likheter mellan CLR och virtuella maskiner.

1.1.1 Klassladdaren

Klassladdarens uppgift är att finna rätt version av klasser och ladda dem i minnet. För att veta vilka klasser som ska laddas så använder klassladdaren metadata i PE-fil (eller snarare *assembly* – se nedan) som aktuell klass (anropande klass) finns i. Ytterligare en uppgift för klassladdaren är att verifiera att koden i PE-fil är typsäker, dock inte pålitlig (*trusted*). Typsäker kod är kod som CLR hanterar, d.v.s. kod som exekverar i CLR. Pålitlig kod är kod där ursprung kan verifieras och som vi kan, m.h.a. säkerhetsinställningar, ange att vi litar på.

1.1.2 JIT-kompilatorn

EXE- eller DLL-filer skapade för .NET innehåller IL-kod (*Intermediate Language*) som JIT-kompilatorn (*Just-In-Time*) kompilerar till maskinkod när koden anropas. Skälet till detta är

[1] Version 1.0 följer med Visual Studio.NET (v.1.0) och version 1.1 följer med Visual Studio.NET 2003.
[2] Software Development Kit.

för att .NET Framework ska kunna användas på andra plattformar än Windows (maskinkod är hårdvaruspecifik).

JIT-kompilatorn får sitt namn från hur den arbetar – först när en metod anropas så kompileras koden för metoden. Den kompilerade koden (maskinkoden) buffras så länge som processen exekverar.

Om vi inte vill använda JIT-kompilatorn så finns det ett verktyg, NGEN, som kan kompilera koden till maskinkod vid installation av PE-fil.

1.2 Common Base Library (CBL)

CBL innehåller alla standard klasser som ingår i .NET Framework. Dessa utgör grunden för alla andra klasser i .NET (t.ex. ADO.NET) och för de klasser vi själva utvecklar.

1.3 Common Type System (CTS) och Common Language Specification (CLS)

CTS är .NETs gemensamma typsystem, det som gör att klasser i t.ex. C# kan anropas från VB.NET-kod. För att kunna använda kod skrivet i ett språk från ett annat bör vi endast använda de datatyper som fungerar i alla språk, d.v.s. de enkla (eller primtiva) datatyperna, och datatyper vi skapar själv utifrån dessa.

En av nyheterna i VB.NET är att vi nu har stöd för undantagshantering (*exceptions*), vilket gör att vi nu har ett enhetligt sätt att hantera fel i alla .NET-språk.

Observera att det finns en del datatyper, t.ex. de numeriska utan tecken (+ eller -), som endast fungerar i VB.NET men inte tvunget i dom andra .NET-språken.

1.4 .NET Software Development Kit (SDK)

.NET SDK innehåller kompilatorer och annan mjukvara som behövs för att utvecklar program för .NET, d.v.s. vi behöver **inte** Visual Studio.NET (VS.NET) för att utveckla program.

Kompilatorerna i SDK:n är kommandobaserade, men vi kan komplettera med t.ex. IDE:n #develop för att erhålla ett grafiskt gränssnitt.

1.5 ”Lokala” komponenter

En ”lokal” komponent, d.v.s. en som bara används av en användare åt gången (och på en dator), är i .NET en vanlig klass (eller *assembly* om ni vill se komponenter på det viset ☺ – se nästa kapitel för förklaring av *assemblies*). D.v.s. alla klasser (eller *assemblies*) som skapas är komponenter och vi behöver inte längre skapa speciella projekt för att skapa en komponent (som med COM). Det enda vi behöver tänka på är om *assemblies* (komponenter) ska vara privata eller delade. Privata *assembiles* används endast av ett program (ofta då i samma mapp som programmets övriga filer) och delade *assembiles* installeras centralt så att alla program på datorn kan använda dem. Delade *assemblies* måste vara unikt identifierbara (precis som komponenter i COM) och kräver att man använder ett nyckelpar (se *Assemblies och unika namn*).

1.6 Distribuerade komponenter

Distribuerade komponenter är (precis som i t.ex. DCOM/MTS eller J2EE) komponenter som kan anropas från en annan dator. I .NET är vi inte längre beroende av DCOM utan kan använda protokoll som TCP/IP eller HTTP (se kapitel *Remoting – DCOM:s ersättare*).[3]

(Distribuerade komponenter som använder tjänster i MTS/COM+ behandlas i kapitel *Komponenttjänster i .NET*.)

1.7 .NET jämfört med COM/COM+

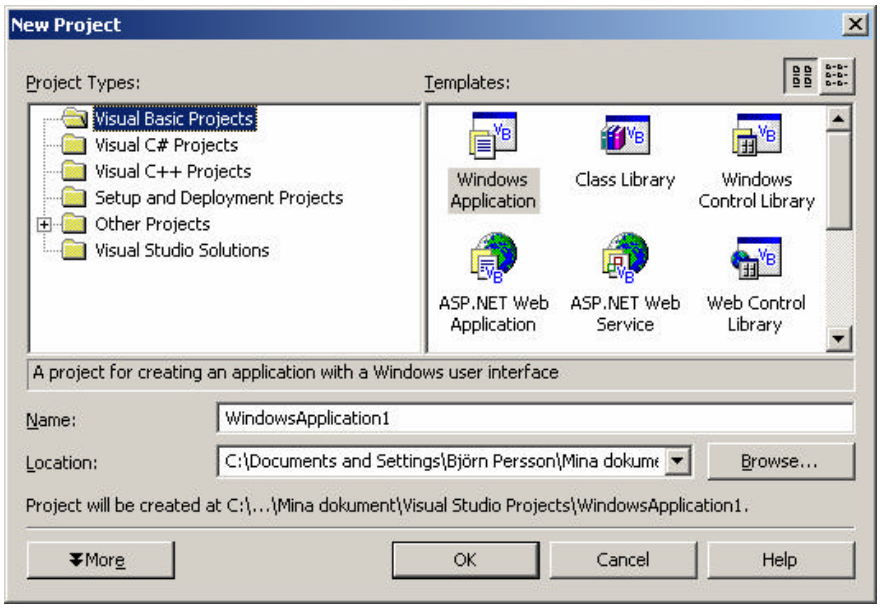
En stor skillnad mellan COM/COM+ och .NET är att alla klasser (eller *assemblies*) i .NET är att betrakta som komponenter. D.v.s. vi behöver inte längre skapa speciella projekt (t.ex. ActiveX DLL i VB6) för att skapa en komponent. Vi har bl.a. CTS att tacka för detta.

En annan viktig detalj är att vi endast kan använda operativsystem som Windows NT4/2000/XP för att utvecklar .NET-applikationer. Vi kan dock (idag!) exekvera programmen även i Windows 98/ME.[4]

1.8 ”Vanliga” projekt i Visual Studio.NET

När vi skapar projekt i Visual Studio.NET (VS.NET) så finns det ett antal att välja mellan. Det är främst tre (men inte enbart!) som är intressant för komponentutveckling:

- *Windows Application* – används för grafiska användargränssnitt (WinForms). Resulterar i en EXE-fil.
- *Class Library* – används för att samla klasser (komponenter[5]). Resulterar i en DLL-fil.
- *ASP.NET Web Application* – används för grafiska, webbaserade användargränssnitt (WebForms). Resulterar i en DLL-fil (på webbserver).



[3] Detta är en sanning med modifikation enligt min mening. Om vi använder MTS/komponenttjänster så är vi fortfarande beroende av DCOM om vi inte skapar vårt eget ”kommunikationsprotokoll” i form av ett stub-objekt (enligt Microsofts sätt att se på stub-objekt, inte Suns!) på servern.
[4] Jag är osäker på om det finns eller kommer finnas någon .NET Framework till Windows 95 – det verkar inte så…
[5] … om vi ser publika klasser som komponenter. Vi kan även se DLL-filen som en komponent…

Alla .NET-språk (de som levereras med VS.NET och de från andra tillverkare) har i stort sett samma typer av projekt. De projekt som beskrivs ovan gäller i alla fall för VB.NET och C# - projekttyperna heter lika och har samma funktioner. Ett projekt innehåller främst ett språk.

Första projektet vi skapar skapar även en lösning[6] (*solution*). En lösning kan innehålla flera projekt, d.v.s. vi behöver inte skapa en ny lösning för varje projekt. För att infoga ett nytt projekt i en lösning så kan vi välja **New**->**Project…** från File-menyn (eller högerklicka på lösningen i Solution Explorer och välja **Add**->**New Project…**). Vill vi blanda språk i en applikation så kan vi skapa ett projekt för respektive språk.

Ett av projekten kan vara det som körs (startprojekt) när vi väljer **Start** från Debug-menyn (eller trycker F5 på tangentbordet). Startupprojektets namn har fet stil i Solution Explorer. För att ange startprojekt så kan vi högerklicka på projektet i Solution Explorer och välja **Set as Startup Project**.

[6] En lösning motsvarar en projektgrupp (*project group*, .VBG-fil) i Visual Basic 6.

2 Assemblies

En ***assembly*** är en modul som vi används för distribution av våra klasser – en modul kan bestå av en (eller flera) EXE- eller DLL-fil(-er). Vår *assembly* innehåller bl.a. klasserna vi skapar men även metadata om vår modul, t.ex. vilka typer som finns i *assembly* samt eventuella referenser till andra *assemblies*. Metadata om en assembly kan placeras i en PE-fil eller i en extern fil. Mest vanligt är att metadata placeras i PE-fil.

När vi skapar projekt i VS.NET så motsvaras (oftast) ett projekt av en *assembly* bestående av en PE-fil. En *assembly* kan vara en del av ett namnutrymme, d.v.s. ett namnutrymme kan bestå av flera *assemblies*.

Om vi skapar ett klassbibliotek (en DLL-fil med klasser i) och sen ett grafiskt gränssnitt (en EXE-fil) så skapar vi alltså två projekt och *assemblies* – en för varje fil.

2.1 Metadata

Metadata i en *assembly* beskriver vad en *assembly* innehåller, d.v.s. typer, och eventuella andra *assemblies* som refereras till. Denna meta data används bl.a. av klassladdaren för att veta vilka klasser som behöver laddas. Ibland kallas metadata för manifest. Metadata genereras utifrån typer (klasser) i *assembly* men kan också påverkas genom att använda attribut (motsvarigheten till egenskaper [*properties*] i VB6 – se *Attributbaserad programmering* i senare kapitel).

Genom att *assemblies* innehåller metadata så är inte .NET längre beroende av systemregistret (eller bara registret för kort) i Windows för att lagra information. Detta leder bl.a. till att .NET inte är lika beroende av operativsystemet, d.v.s. .NET Framework kan användas på andra operativsystem än Windows.

Vi kan dock säga att registrets till viss del har ersatts av Global Assembly Cache (GAC), dock bara för att lagra information om var *assemblies* finns och vilka versioner av assembly som finns. Och om vi exponerar COM-gränssnitt från vår *assembly* så måste dessa gränssnitt registreras i registret för att vara åtkomligt i COM.

2.2 Privata och delade *assemblies*

En *assembly* kan vara privat – används endast av ett program – eller delad – registreras då i *Global Assembly Cache* (GAC) för att kunna användas av flera program. DLL-filer som är privata *assemblies* bruka placeras i samma mapp som klientens *assembly* (t.ex. EXE-fil med grafiskt gränssnitt). En delad *assembly* **kan** även användas som en privat, d.v.s. den behöver inte installeras i GAC.

2.2.1 Referenser till *assemblies*

När vi anger en referens till en privat *assembly*, t.ex. en DLL-fil, så anger vi sökvägen till en fil. Om denna sökväg ändras efter att applikation kompilerats så genererar CLR:en ett felmeddelande. För att undvika detta kan vi t.ex. kopiera externa *assemblies* till samma mapp som *assembly* vi håller på att skapa. Därmed kan vi kopiera alla PE-filer i mappen samtidigt för att installera på annan dator.

När vi anger referenser till delade *assemblies* så anger vi endast att vi vill använda vald *assembly*, d.v.s. inte sökväg till DLL-filer. För att kunna ange referenser till en delad *assembly* måste vi ha tillgång till DLL-filen. Men för att kunna exekvera vår *assembly* måste dock alla delade *assemblies* (som vi satt referenser till) vara installerade i GAC. D.v.s. efter att en delad

assembly har installerats i GAC, och alla referenser har satts till delad *assembly*, så behöver vi inte DLL-filen för delad *assembly*.

2.2.2 Assemblies och unika namn

En delad *assembly* måste (i motsats till en privat *assembly*) vara garanterat unik (precis som våra komponenter i COM) och för det använder vi publika och privata nycklar (*public and private keys*). Vi behöver endast skapa en uppsättning av nycklar för alla *assemblies* vi skapar, d.v.s. inte en ny uppsättning för varje *assembly* som vi skapar.

Den privata nyckeln används för att signera en *assembly*, vilket bl.a. gör att man kan verifiera skaparens identitet (ett skäl till att endast generera en uppsättning av nycklarna). Detta gör att andra t.ex. kan använda våra nycklar för att endast exekvera kod som dom litar på.

För att generera publika nycklar så kan vi använda verktyget `SN.EXE` (*Strong Name utility*) (se exempel med delad *assembly* nedan).

2.2.3 Versionshantering

I motsats till COM så kan .NET hantera flera versioner av samma komponent. D.v.s. vi kan ha klienter som använder olika versioner av samma komponent. Versionshantering är dock främst intressant då vi talar om delade *assemblies*.

När vi skapar projekt i VS.NET så skapas en fil `AssemblyInfo.vb` (eller `.cs` ☺) där det bl.a. finns attribut för just version på vår *assembly*.

2.3 Registrera delad assembly i Global Assembly Cache (GAC)

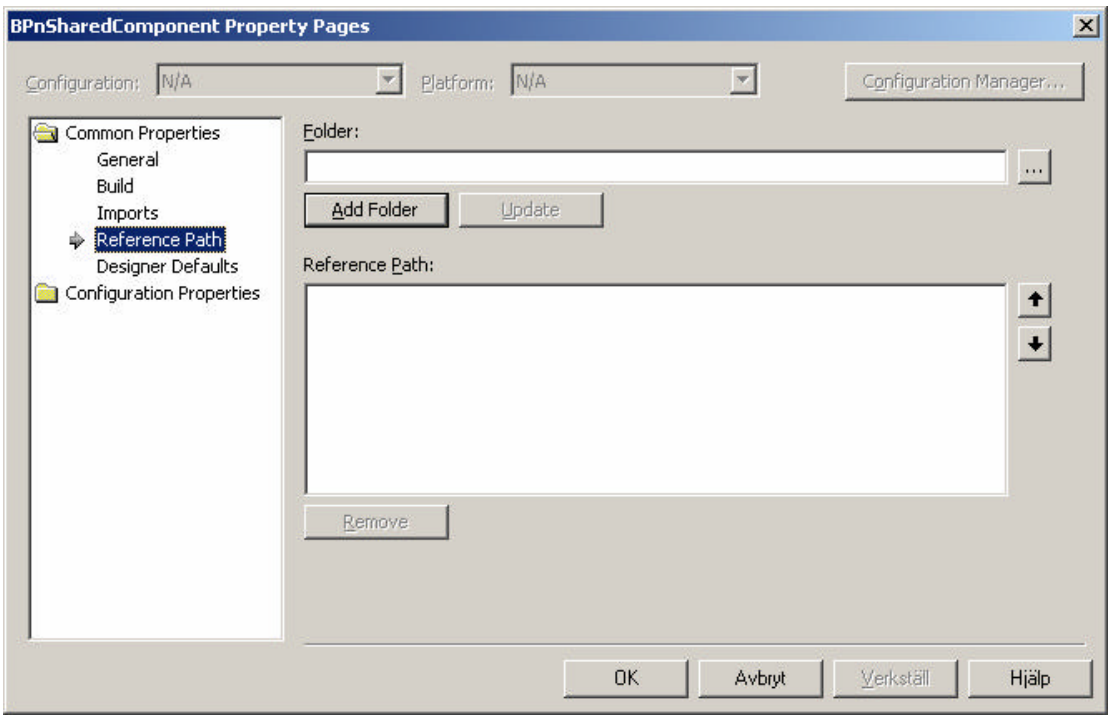
Det finns två sätt att registrera en delad *assembly*: *drag-and-drop* eller med verktyget `GACUTIL.EXE`. Lättast är givetvis att använda *drag-and-drop* – vi drar DLL-filen och släpper den i mappen `%WINDIR%\Assembly`[7].

2.3.1 Visa egna delade *assemblies* i dialogrutan Add Reference i VS.NET

Observera att delade *assemblies* **inte** hamnar i dialogrutan Add Reference bara för att dom registreras i GAC. Vill vi, på ett relativt enkelt sätt, kunna använda dialogrutan Add Reference för att ange en referens till våra delade *assemblies* bör vi göra följande:

1. Skapa en mapp (lämpligen under `C:\Program\` [8]) att placera våra DLL:er i.
2. Högerklicka på projektet där vi vill använda egen utvecklade *assemblies* i och välj **Properties**. Dialogrutan <Projektnamn> Property Pages visas (se bild nedan).

[7] Mappen `%WINDIR%` är den mapp som Windows installerats i, t.ex. `C:\Windows` om Windows XP eller `C:\WinNT` om Windows NT 4.
[8] Observera att ni kanske inte har rättigheter till denna mapp om ni inte är administratör på datorn!



3. Expandera **Common Properties** och klicka på **Reference Path**.
4. I textrutan Folder – fyll i sökvägen till mapp (eller bläddra till mapp genom att klicka på knappen med tre punkter till höger om textruta) som skapades i punkt 1 och klicka på **Add Folder**.
5. Klicka på **OK** för att stänga dialogrutan <Projektnamn> Property Pages.

2.4 Skapa och använda en privat assembly

I detta exempel skapar vi först ett projekt av typen Class Library och sen en klient (projekt av typen Windows Application) för att testa vår assembly. Projekten kommer skapas i separat ”lösningar” (*solutions*) för att visa att komponenten är skild från klienten. Men vi kunde lika gärna skapa ett nytt projekt (eller snarare en lösning, *solution*) för klienten och sen lägga till vår privata assembly som ytterligare ett projekt i samma lösning.[9]

2.4.1 Skapa privat assembly

1. Skapa ett nytt projekt av typen **Class Library** med namnet `BPnPrivateComponent` (ersätt gärna prefixet ”BPn” med er användaridentitet i nätverket) och klicka på OK.
2. Byt namn på klassen (från `Class1` till `Hello`) samt på filen (som ni antingen kan byta i Solution Explorer eller Properties).
3. Lägg till metoden `SayHello` genom att fylla i koden för metoden i klassen. Fullständig kod för klassen visas nedan.

```
Public Class Hello

    'Metod som returnerar en sträng med bl.a. datum och tid
    Public Function SayHello() As String
```

[9] Genom att börja med att skapa klienten först så kommer den att exekveras när vi trycker på F5-tangenten.

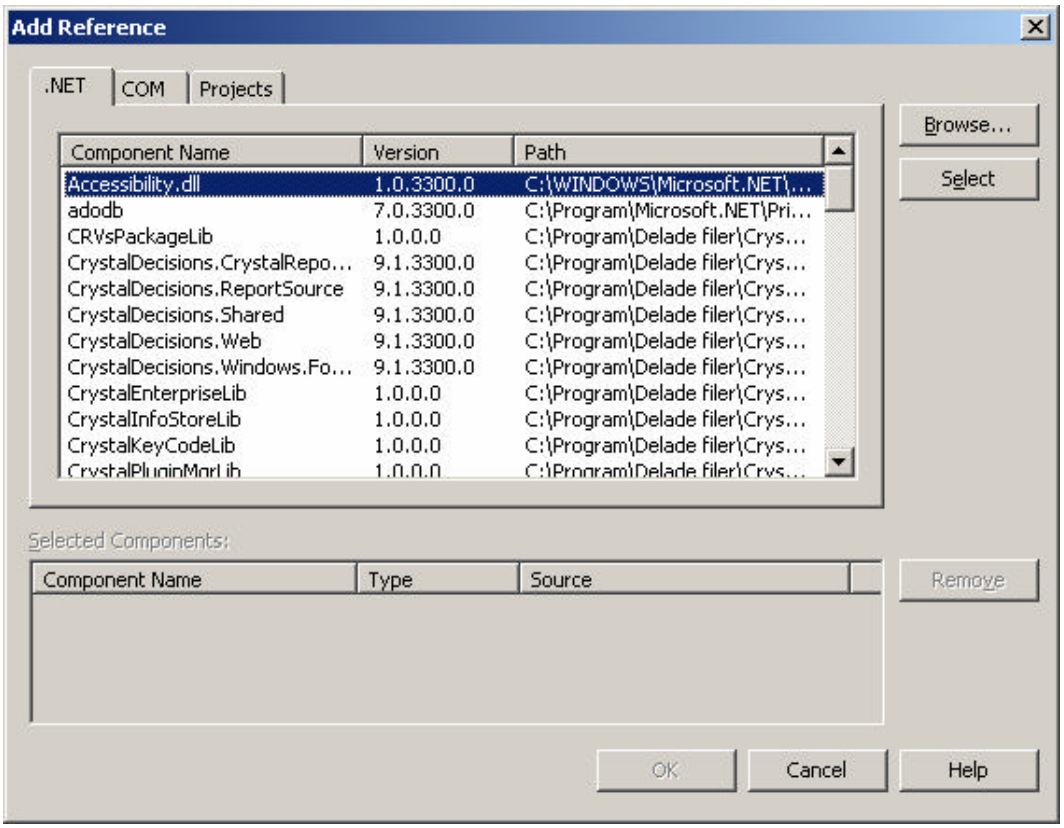
```
            Return "Hej, datum och tid är " & DateTime.Now
        End Function

    End Class
```

4. Kompilera projektet genom att välja **Build Solution** från Build-menyn (eller trycka Shift-Ctrl-B på tangentbordet). Klart!

2.4.2 Skapa klient för privat assembly

1. Skapa ett nytt projekt av typen **Windows Application** med namnet `BPnPrivateClient` (ersätt gärna prefixet ”BPn” med er användaridentitet i nätverket) och klicka på OK.
2. Lägg till en referens till privat assembly som skapades ovan. Detta kan bl.a. göras genom att högerklicka på projektet (inte lösningen – *solution*!) i Solution Explorer och välja Add Reference… – dialogrutan Add Reference visas (se bild nedan).



Klicka på knappen Browse i dialogrutan och bläddra dig fram till mappen med DLL-filen för privat *assembly* (som skapades i förra avsnittet). DLL-filen finns i BIN under projektmappen för vår privata *assembly*. Markera DLL-filen och klicka på OK och sen OK igen (för att stänga dialogrutan Add Reference).

3. Dra och släpp en knapp (av typen Button) på formuläret. (Ändra egenskaperna **(Name)** och **Text** om ni så önskar.)
4. Dubbelklicka på knappen för att lägga till händelsehanterare för knappen (`Button1_Click()`) och lägg till de tre raderna kod för metoden (proceduren). Nedan visas fullständig kod för formulärets klass (med automatgenererad kod dold – ”textrutan” med texten *Windows Form Designer genererated code* i).

```
Public Class Form1
    Inherits System.Windows.Forms.Form

 Windows Form Designer generated code

    'Metod som anropas när användare klicka på knapp
    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) Handles Button1.Click

        Dim objHello As BPnPrivateComponent.Hello   'Deklarera variabel
        objHello = New BPnPrivateComponent.Hello()  'Skapa instans av klass
        MsgBox(objHello.SayHello)                   'Anropa metod i instans

    End Sub

End Class
```

5. Kompilera och kör programmet. Klicka på knappen för att anropa metoden i den ”lokala” privata komponenten.

Om vi tittar i mappen `BIN` (under klientens projektmapp) så ser vi att DLL-filen för vår privata *assembly* har kopierats till denna mapp av VS.NET. För att installera detta program på en annan dator behöver vi alltså endast kopiera innehållet i mappen `BIN` till den andra datorn.

2.5 Skapa och använda en delad assembly

Skillnaden mellan en privat och en delad assembly är som sagt att en delad *assembly* även har ett unikt namn (*strong name*). För detta behöver vi alltså skapa en nyckelfil med verktyget `SN.EXE`.

Stegen för att skapa detta exempel är i många fall de samma som när vi skapade en privat *assembly*. Och koden som vi kommer skriva själva är nästan identisk med den för vår privata *assembly* (som skapats ovan).

2.5.1 Skapa en delad assembly

1. Skapa ett nytt projekt av typen **Class Library** med namnet `BPnSharedComponent` (ersätt gärna prefixet ”BPn” med er användaridentitet i nätverket) och klicka på OK.
2. Byt namn på klassen (från `Class1` till `Hello`) samt på filen (som ni antingen kan byta i Solution Explorer eller Properties).
3. Lägg till metoden `SayHello` genom att fylla i koden för metoden i klassen. Fullständig kod för klassen visas nedan.

```
Public Class Hello

    'Metod som returnerar en sträng med bl.a. datum och tid
    Public Function SayHello() As String
        Return "Hej, datum och tid är " & DateTime.Now
    End Function

End Class
```

4. Skapa en nyckelfil genom att skriva ”`SN.EXE -k bpn.snk`”[10] i kommandotolken.[11] Placera nyckelfilen i en ”lämplig” mapp (d.v.s. en som är lätt att skriva sökvägen till ☺) – vi behöver filen varje gång vi ska kompilera projektet. Vi kan använda denna nyckelfil till alla delade komponenter vi skapar i framtiden om vi vill.
5. Öppna filen `AssemblyInfo.vb`, genom att t.ex. dubbelklicka på den i Solution Explorer, och lägg till nedanstående rad (sist eller tillsammans med liknande attribut). (Ändra sökvägen för att matcha sökvägen till den mapp där ni la er nyckelfil.)

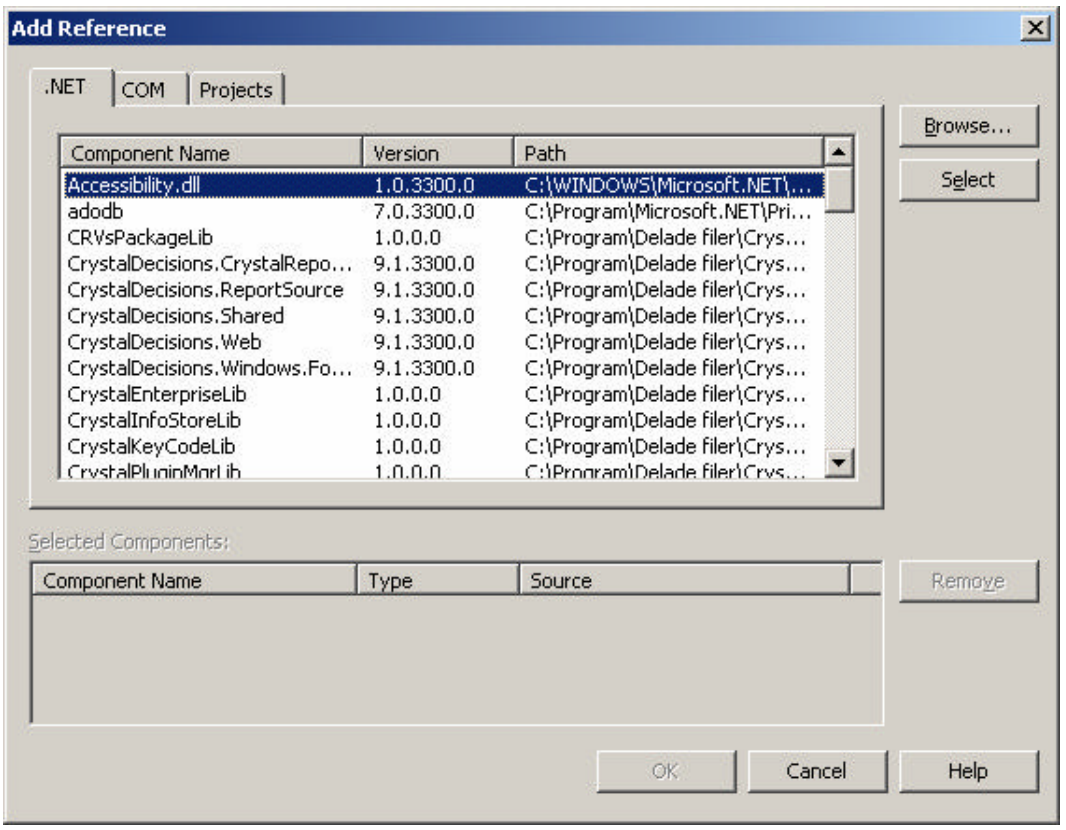
```
<Assembly: AssemblyKeyFile("C:\Student\bpn.snk")>
```

6. Kompilera projektet genom att välja **Build Solution** från Build-menyn (eller trycka Shift-Ctrl-B på tangentbordet).
7. Registrera assembly i GAC genom att skriva ”`gacutil /i bpnsharedcomponent.dll`”[12] i kommandotolken (eller dra och släpp DLL-filen i mappen `%WINDIR%\Assembly`). Observera att detta steg kan kräva att ni har administratörsrättigheter på datorn!
8. Klart!

2.5.2 Skapa klient för delad assembly

1. Skapa ett nytt projekt av typen **Windows Application** med namnet `BPnSharedClient` (ersätt gärna prefixet ”BPn” med er användaridentitet i nätverket) och klicka på OK.
2. Lägg till en referens till delad assembly som skapades ovan. Detta kan bl.a. göras genom att högerklicka på projektet (inte lösningen – *solution*!) i Solution Explorer och välja Add Reference… – dialogrutan Add Reference visas (se bild nedan).

[10] Filen `SN.EXE` finns antagligen i mappen `C:\Program\Microsoft Visual Studio .NET\FrameworkSDK\Bin\` om VS.NET installerades samtidigt med .NET Framework. D.v.s. vi måste lägga till sökvägen till mappen före `sn.exe` om det inte fungerar att bara skriva `sn.exe`. Observera mellanslaget före ”`.NET`”!
[11] Ni kan ersätta bpn.snk med det namn ni vill att nyckelfilen ska ha – filändelsen .SNK är dock ”standard”.
[12] Filen `GACUTIL.EXE` finns antagligen i mappen `%WINDIR%\Microsoft.NET\Framework\v1.0.3705\` (beroende på version av .NET Framework).



Klicka på knappen **Browse** i dialogrutan och bläddra dig fram till mappen BIN under projektmappen för DLL-filen för delad assembly (som skapades i förra avsnittet). Markera filen och klicka på **OK** och sen **OK** igen (för att stänga dialogrutan Add Reference). Vi behöver alltså tillgång till DLL-fil för att utveckla klienter till *assemblies* – vid exekvering räcker det med att DLL-fil är installerad i GAC och vi kan slänga DLL-filen.

3. Dra och släpp en knapp (av typen Button) på formuläret. (Ändra egenskaperna **(Name)** och **Text** om ni så önskar.)
4. Dubbelklicka på knappen för att lägga till händelsehanterare för knappen (`Button1_Click()`) och lägg till de tre raderna kod för metoden (proceduren). Nedan visas fullständig kod för formulärets klass (med automatgenererad kod dold – ”textrutan” med texten *Windows Form Designer genererated code* i).

```
Public Class Form1
    Inherits System.Windows.Forms.Form

  Windows Form Designer generated code

    'Metod som anropas när användare klicka på knapp
    Private Sub Button1_Click(ByVal sender As System.Object, _
                              ByVal e As System.EventArgs) Handles Button1.Click

        Dim objHello As BPnSharedComponent.Hello    'Deklarera variabel
        objHello = New BPnSharedComponent.Hello()   'Skapa instans av klass
        MsgBox(objHello.SayHello)                   'Anropa metod i instans

    End Sub

End Class
```

5. Kompilera och kör programmet. Klicka på knappen för att anropa metoden i den ”lokala” delade komponenten.

Detta exempel visar att användningen av privata och delade assemblies fungerar på samma sätt. Skillnaden ligger i att DLL-filen för en delad assembly inte kopieras till mappen BIN för klienten. För att installera detta program på en annan dator så måste vi alltså kopiera mappen BIN (i projektmappen för klienten) och DLL-filen med vår delade assembly samt registrera vår delade assembly i GAC på den andra datorn.

3 *Remoting* – DCOM:s ersättare

Distribuerade objekt i COM använder DCOM[13] för kommunikation över nätverk medan de kan nås via *remoting* i .NET. Skillnaden mellan *remoting* och DCOM är att vi kan använda oss inte bara av DCOM som kommunikationsprotokoll, utan även av andra standardiserade protokoll som TCP/IP och HTTP. I.o.m. detta så behöver inte brandväggar på Internet vara ett problem längre.[14]

Remoting är även lättare då vi inte behöver gör en massa inställningar på server och klient för att använda *remoting*. Men vill vi använda *remoting* (istället för DCOM) med komponenter installerade i MTS/komponenttjänster så krävs dock lite merarbete.

3.1 Klasser och gränssnitt i *remoting*

Distribuerade objekt i .NET måste ärva, direkt eller indirekt, från klassen `MarshalByRefObject` (motsvaras av gränssnittet `java.rmi.Remote` i Java). Denna klass innehåller många av de egenskaper som krävs av ett distribuerat objekt – d.v.s. mer kod som vi inte behöver skriva (återigen kan vi fokusera på affärslogiken ☺). Distribuerade objekt är, vad man kallar, typen *marshal by reference*.

Klasser som inte ska användas distribuerat, men som kan skickas som parametrar till distribuerade objekts metoder eller som returvärden från metoderna, sägs vara av typen *marshal by value*. Dessa klasser måste märkas med attributet `Serializable`. (Se avsnittet *Attributbaserad programmering* nedan för mer information om attribut.)

```
<Serializable> _
Public Class ParameterKlass
  'Implementation av klass...
End Class 'ParameterKlass
```

För att erhålla en referens till ett distribuerat objekt så behöv någon form av mekanism för att hitta objekten. Denna mekanism heter kanaler i .NET. För att göra ett distribuerat objekt tillgängligt så registreras objektet i en kanal. Klienter kan sen fråga kanalen om en referens till objektet. Som standard levereras .NET med två typer av kanaler: HTTP och TCP. Dessa kanaler motsvaras av klasserna HttpChannel respektive TcpChannel. Endast en kanal av respektive typ kan registreras per dator.

Distribuerade objekt vars referens returneras via ett returvärde (eller som en utparameter) från en metod behöver inte registreras i en kanal.[15] Detta sker med automatik av .NET (en s.k. ObjRef med information om kanaler, m.m. returneras till klient – se MSDN för mer information). (Detta kan användas av designmönster så som Factory där vi använder en metod, i en viss klass, för att skapa/returnera en instans av en annan klass.) Vad jag vill komma fram till är att alla distribuerade objekt **inte** behöver vara registrerade. Endast det ”första” objektet behöver vara registrerat – referenser till resterande objekt kan fås genom att returnera dem från ”första” objektets metoder. Alla distribuerade objekt **måste** dock ärva från klassen `MarshalByRefObject` för att vara distribuerade.

[13] Distributed COM.
[14] Brandväggar används för att förhindra oönskad nätverkstrafik in eller ut från ett privat nätverk. Vanligtvis är endast trafik för kända protokoll (t.ex. HTTP) tillåtet, vilket förhindrar protokoll som DCOM att fungera.
[15] Barnaby föreslår namnet *marshaled object* för denna typ av distribuerade objekt.

(Distribuerade objekt i .NET påminner mycket om de i J2EE/RMI. I motsatts till Java RMI så använder vi dock ingen namnserver, RMI-registry.)

3.2 Sätt att registrera distribuerade objekt i Remoting

Det finns främst två sätt att registrera distribuerade objekt i Remoting: förregistrera och klientaktivera. För att förregistrera ett objekt skapas av någon form av server som skapar det distribuerade objektet. När objekt förregistreras så används metoden `RegisterWellKnownServiceType()` i klassen RemotingConfiguration. Konstanten `WellKnownObjectMode.Singleton` anger att endast ett objekt ska hantera alla klienters anrop och konstanten `WellKnownObjectMode.SingleCall` att ett objekt för varje klient ska skapas. Observera att objekt i det senare fallet **inte** behåller tillstånd mellan anrop![16]

När objekt registreras för att vara klientaktiverade så använder vi endast server för att registrera typen i kanalen. Instanser av klassen skapas först när klienter begär en instans. När en klient begär ett objekt så skapas en instans av klassen, klient anropar metod i instansen och sen förstörs instansen igen. För klientaktiverade objekt används metoden `RegisterActivatedServiceType()` i klassen RemotingConfiguration vid registrering. (Klientaktiverade objekt behandlas inte mer i denna sammafattning.)

3.3 Exempel på distribuerat objekt

I exempel nedan kommer vi att använda TCP som kommunikationsprotokoll, men koden för t.ex. HTTP skulle se ungefär likadan ut – ersätt TcpChannel med HttpChannel.

Här skapas först det distribuerade objektet (eller klassen för den – `MittObjekt`) och sen servern (`MinServerApp`) som registrerar en instans av klassen. Båda dessa (d.v.s. dess *assemblies*) placeras på datorn som det distribuerade objektet ska exekvera på.

Sen skapar vi en klientapplikation (`MinKlient`) som begär en referens till det distribuerade objektet. Klienten **och** *assembly* med det distribuerade objektet måste placeras på datorn där klienten ska exekvera. Detta då klienten anropar metoder i det distribuerade objektet och därmed behöver klienten känna till vilka metoder (klassens publika gränssnitt) som finns i det distribuerade objektet.[17]

Innan vi försöker köra exemplet på två datorer bör vi testa det på samma dator. Glöm inte att det är lättare att testa saker i en enkel miljö innan vi blandar in mer komplexa saker (så som nätverkskommunikation). Vi bör alltså testa klassen för det distribuerade objektet direkt innan vi börjar blanda in servern som registrerar det i en kanal.

3.3.1 Distribuerat objekt

I detta exempel skapas ett enkelt objekt (eller klass om vi ska vara exakta ☺). Jag har gjort valet att inte göra objektet för avancerat för att visa att distribuerade objekt inte skiljer sig så mycket från vanliga objekt (eller snarare klasser). För att visa på vad som händer så görs utskrifter till kommandotolken när t.ex. konstruktor eller metod anropas.

För att göra objektet (klassen) distribuerat så måste vår klass ärva från `MarshalByRefObject`. Annars är det inget ”speciellt” med denna klass. Varken konstruktor eller destruktor gör något

[16] Om sanningen ska fram så skapas ett objekt för att hantera ett metodanrop från klient för att förstöras direkt efter att metod exekverat färdigt. Därför behålls inget tillstånd.
[17] Om vi använder VS.NET och sätter en referens till assembly med det distribuerade objektet så kommer VS.NET kopiera assemblyn till samma mapp som assembly för klienten. D.v.s. när vi ska kopiera filerna till klienten så kopierar vi bägge samtidigt.

konstruktivt och metoden `MinMetod()` returnerar en sträng med bl.a. tiden (på dator där objektet exekverar).

```
Public Class MittObjekt
  Inherits MarshalByRefObject

  'Konstruktor
  Public Sub New()
    Console.WriteLine("Objekt skapas...")
  End Sub

  'Destruktor
  Protected Overrides Sub Finalize()
    Console.WriteLine("Objektet förstörs...")
  End Sub

  'Metod att anropa
  Public Function MinMetod() As String
    Console.WriteLine("Metod har anropats...")
    Return "Hej, tiden är nu " & DateTime.Now    'Returnera sträng med aktuell tid
  End Sub
End Class
```

3.3.2 Server som skapar en instans av distribuerat objekt

Här har koden för att registrera det distribuerade objektet lagts i en separat modul. Men Main-metoden skulle lika gärna ha placerats i klassen för vårt distribuerade objekt ovan (och vi hade haft färre klasser att jobba med). Å andra sidan så kommer inte alla klasser att registrera sig – en del kommer t.ex. returneras från metoder i objekt som registrerats.

Innan vi skriver koden så måste vi sätta en referens till System.Runtime.Remoting för projektet. Sen importerar vi tre namnutrymmen med `Imports` (se kod nedan).

För att kunna göra vårt distribuerade objekt tillgängligt för klienter utanför datorn behöver vi en kanal. I detta exempel skapas en kanal för protokollet TCP med port 4000 och som sen registreras. Sist av allt registrera vi objektet med adressen ”MinServer”.

```
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Tcp

Module MinServerApp

  Sub Main()
    Dim kanal As TcpChannel

    Console.WriteLine("Server startars...")
    Console.WriteLine("Tryck ENTER för att avsluta server...")

    'Skapa en kanal att anropa objekt på
    kanal = New TcpChannel(4000)

    'Registrera kanalen för att göra den tillgänglig
    ChannelServices.RegisterChannel(kanal)

    Console.WriteLine("Skapar och registrerar objekt...")

    'Registrera adress till objekt
    RemotingConfiguration.RegisterWellKnownServiceType( _
    GetType(MittObjekt.MittObjekt), "MinAdress", WellKnownObjectMode.Singleton)

    Console.ReadLine()  'Vänta på ENTER
    Console.WriteLine("Server avslutas...")
  End Sub
End Module
```

3.3.3 Klient till distribuerat objekt

Jag har valt att göra klienten kommandobaserad (för exekvering i kommandotolken) för att inte göra koden för lång. Koden i `Main()` kan givetvis placeras i en händelsehanterare för (”bakom en”) knapp.

Innan vi skriver koden så måste vi sätta en referens till System.Runtime.Remoting för projektet.

Precis som i servern så måste vi skapa en kanal – här behöver vi dock inte tala om vilken port då vi gör det när vi anger URL[18] till objektet. Det ”distribuerade” i klienten sker m.h.a. den statiska metoden `GetObject()` i klassen Activator. Här hämtas en referens till det distribuerade objektet.

När vi väl hämtat en referens till objektet så anropar vi metoder i objektet som om det vore ett lokalt objekt (dock med lite längre svarstider om vi måste skicka anrop över nätverk).

```
'Importera namnutrymmen för att slippa "långa namn" i övrig kod
Imports System.Runtime.Remoting
Imports System.Runtime.Remoting.Channels
Imports System.Runtime.Remoting.Channels.Tcp

Module MinKlient
  Private kanal As TcpChannel
  Private objMitt As MittObjekt.MittObjekt

  Sub Main()
    Dim strText as String

    Try
      'Skapa en kanal
      kanal = New TcpChannel()

      'Registrera kanalen
      ChannelServices.RegisterChannel(kanal)

      Console.WriteLine("Skapar objekt (på annan dator)")

      'Skapa ett objekt - ändra adress till dator (om annan än lokal)
      objMitt = CType( _
      Activator.GetObject(GetType(MittObjekt.MittObjekt), _
         "tcp://127.0.0.1:4000/MinAdress"), MittObjekt.MittObjekt)

      Console.WriteLine("Anropar metod i objekt")

      'Anropa metod i distribuerat objekt
      strText = objMitt.MinMetod()
      Console.WriteLine(strText)
    Catch ex As Exception
      Console.WriteLine(ex.Message)
    End Try

    Console.WriteLine("Tryck ENTER för att avsluta...")
    Console.ReadLine()
  End Sub
End Module
```

[18] URL = Uniform Resorce Locator – en standard för hur adresser ges.

3.3.4 En mer praktisk lösning…

En mer praktisk lösning är att implementera servern som en tjänst (*Windows Service*) så att vi inte behöver ha ett kommandotolksfönster öppet. En tjänst exekverar i bakgrunden och kan t.ex. startas då Windows startas.

3.3.5 Några fler saker att tänka på

Som exempel ovan utvecklats så måste hela klassen för det distribuerade objektet finnas tillgängligt på klienten. En bättre lösning är att skapa ett gränssnitt och låta det distribuerade objektet implementera gränssnittet. På så sätt behöver endast gränssnittet finnas på klienten och klassen som implementerar gränssnittet finns endast på servern.

(Denna sida har avsiktligt lämnats tom.)

4 Komponenttjänster i .NET [UTVECKLA]

.NET Framework är systemmjukvara som installeras ovanpå operativsystemet (d.v.s. mellan mjukvaran vi utvecklar och operativsystemet). Detta innebär att vi i .NET kan använda oss av de tjänster som operativsystem (och tilläggspaket som Windows NT 4 Option Pack med Microsoft Transaction Server, MTS) ger. .NET är alltså inte bara nytt! Konsekvenserna av detta är att de begränsningar som vi har i COM/COM+ till viss del även finns i .NET.[19] Våra .NET-komponenter som använder sig av MTS/COM+ tjänster (kallat ***Enterprise Services*** i .NET) kan fortfarande inte behålla sitt tillstånd (d.v.s. värden i instansvariabler) mellan metodanrop.

När vi vill utveckla distribuerade applikationer så använder vi alltså fortfarande de tjänster som MTS och komponenttjänster (*Component Services*, COM+) ger i Windows NT 4 respektive Windows 2000/XP[20]. En skillnad är dock att vi inte är hänvisade till att bara använda DCOM (*Distributed COM*) för kommunikation mellan klient och server – vi kan även använda t.ex. HTTP som protokoll. Men om vi vill använda MTS/COM+ tjänster och ett annat protokoll än DCOM så måste vi dock göra lite mer jobb för att det ska fungera.

För att använda komponenttjänsterna så måste vi använda klasser och attribut i namnutrymmet `System.EnterpriseServices`. Vi måste alltså sätta en referens till detta namnutrymme och, för att minska mängden kod (d.v.s. slippa upprepa namnutrymmet hela tiden), så kan vi (om vi vill) även importera namnutrymmet. Klasserna som ska använda komponenttjänsterna måste sen ärva från klassen `ServicedComponent` (i nämnt namnutrymme).[21] Genom att ärva från denna klass så blir klassen en distribuerad klass, d.v.s. kan användas mellan processer/över nätverk.

Våra projekt måste även vara delade *assemblies*, eller snarare de måste ha ett *strong name*. Vi måste alltså använda en nyckelfil (med ett nyckelpar i), men vi behöver inte installera dem i GAC för att de ska fungera. Det är dock att rekommendera att installera dem i GAC.

I detta kapitel beskrivs hur vi kan använda komponenttjänster i .NET och vad som krävs för att göra det. Nästa kapitel beskriver mer praktiskt hur man gör (med det ”klassiska” bokningssystemet för datorer ☺).

4.1 Attributbaserad programmering

I VB6 använder man bl.a. egenskaper för att ange vilka tjänster, t.ex. transaktioner, som en komponent ska använda. Men i VB.NET använder vi attribut istället. Attribut kan appliceras på hela *assemblies*, klasser eller metoder. (För mer om kategorisering av attribut – sök på *attributes [Visual Basic], overview* i MSDN.) I VB.NET använder vi ”hakparenteserna” < och > för att innesluta ett attribut samt ”fyrkantsparenteserna [och] i C#.[22]

```
<Attributnamn(Parameter1, Parameter2, ...)>   'VB.NET

[Attributnamn(Parameter1, Parameter2, ...)]   //C#
```

Om fler än ett attribut ska appliceras så använder vi endast ett ”hakparentespar” och skiljer attributen åt med kommatecken (”,”).

[19] … vilket är ett av skälen till varför det är bra att lära sig COM/COM+ först. ☺
[20] Jag borde egentligen lägga till även Windows 2003 Server här…. ☺
[21] `ServicedComponent` ärver från `ContextBoundObject`, som i sin tur ärver från `MarshalByRefObject`.
[22] Även i VC++.NET används fyrkantiga hakparenteser ([och]).

```
<Attributnamn1(Parameter1), Attributnamn2(Parameter2)>   'VB.NET
```

Attribut som ska appliceras på en *assembly* placeras oftast i filen `AssemblyInfo.vb`[23] för att samla dem på ett ställe. Dessa attribut föregås även av det reserverade ordet `Assembly`.

```
<Assembly: Attributnamn1(Parameter1)>   'VB.NET
```

Attributen som ska appliceras på klasser eller metoder placeras före respektive klass/metod.

```
<Attributnamn1(Parameter1)> _
Public Class MinKlass                    'VB.NET
  'Implementation av klass...
End Class
```

Observera att attribut måste skrivas på samma rad som den klass eller metod som attributet ska appliceras på! Det går, i VB.NET, att använda ”fortsättningstecknet” (*underscore*, ”_”) för att göra koden lite mer lättläst (som i exempel ovan). I C# avslutats satser med ett semikolon, d.v.s. vi behöver inte tala om att sats fortsätter på nästa rad.

```
[Obsolete("Denna metod används inte längre")]
public void GammalMetod()
{
  //Implementation för metod
}
```

Attribut använder vi bl.a. för att ange en beskrivningar, exportera gränssnittet `IDispatch` (i COM) och för hur transaktioner ska hanteras.

4.1.1 Typer av attribut

Några attribut för MTS/COM+ (i namnutrymmet `System.EnterpriseServices`) visas i tabell nedan. (För en komplett lista på fördefinierade attribut, sök på *Attribute class, about Attribute class* i MSDN och klicka på länken *Derived classes.*)

Typ	Namn	Standardvärde	Beskrivning
Assembly	ApplicationActivation	Library	Hur COM+-tillämpning ska aktiveras (server eller library)
	ApplicationID	Genererat GUID	GUID för COM+-tillämpning
	ApplicationName	Namn på assembly	Namn på COM+-tillämpning
Klass	JustInTimeActivation	False	Om JIT-aktivering ska användas
	ObjectPooling	False	Om objekt ska placeras i objektpool
	Synchronization	False	Om COM+ ska hantera synkronisering
	Transaction	False	Om COM+ ska hantera transaktioner
Metod	AutoComplete	-	Om metod ska anropa SetComplete() automatiskt

[23] … eller .cs om C#.

4.1.2 Exportera gränssnittet IDispatch mot COM-klienter

Om vi vill använda vår .NET-komponent i t.ex. ASP (inte i ASP.NET) så måste vi även exportera gränssnittet `IDispatch` mot COM.[24] Detta gör vi genom att använda attributet `ClassInterface` som finns i namnutrymmet `System.Runtime.InteropServices`, d.v.s. ytterligare ett namnutrymme att importera.

```
Imports System.EnterpriseServices
Imports System.Runtime.InteropServices

'Exportera gränssnittet IDispatch för COM-klienter
<ClassInterface(ClassInterfaceType.AutoDual)> _
Public Class MinKlass
  Inherits ServicedComponent
  'Implementation av klassen...
End Klass
```

4.2 Förbereda klasser och projekt för Component Services

För att använda komponenttjänster (*Component Services*) i våra komponenter så måste vi bl.a. sätta referenser till några namnutrymmen samt applicera några attribut på projekt, klasser och metoder. Några av attributen ska appliceras på enstaka klasser och metoder samt några ska vi applicera på hela projektet, d.v.s. i filen `AssemblyInfo.vb`. Vi kan också, för att minska mängden gånger vi behöver ange fullständiga namn på klasser (inklusive attributklasser), importera namnutrymmen – både i klasser och i filen `AssemblyInfo.vb`.

4.2.1 Förbereda klasser

Först måste vi sätta en referens till namnutrymmet `System.EnterpriseServices` för vårt projekt, vilket vi gör genom att välja **Add Reference…**. För att göra det lättare för oss (d.v.s. slippa behöva skriva ovan nämnda namnutrymme hela tiden) kan vi också importera samma namnutrymme (se kodexempel nedan). Nästa steg är att låta vår klass ärva från klassen `ServicedComponent` (i nämnda namnutrymme) samt att applicera attributen för de tjänster vi vill ha och om vi vill exportera gränssnittet `IDispatch` för COM-klienter (se kodexempel nedan). Och sist av allt så implementerar vi vår komponent.

```
Imports System.EnterpriseServices

<Attribut_att_applicera()> _
Public Class MinKlass
  Inherits ServicedComponent

  'Implementation av klassen...

End Class
```

4.2.2 Förbereda projekt

För att slippa behöva upprepa vissa attribut (och för att samla på ett stället) så använder vi filen `AssemblyInfo.vb` för gemensamma attribut som gäller för alla klasser i projektet. Bl.a. så använder vi attribut för att tala om var vår nyckelfil finns (se avsnitt *Privata och delade*

[24] Om vi gör detta så slipper vi även en varning när vi registrera komponent i en COM+-applikation ☺.

assemblies), namn på COM+-applikation och vilken typ av applikation (bibliotek- eller serverapplikation) det ska vara.

För att slippa behöva upprepa namnutrymme för attributet så kan vi importera `System.EnterpriseServices` även i filen `AssemblyInfo.vb`.

För namn på COM+-applikation använder vi attributet `ApplicationName` och för typ av applikation attributet `ApplicationActivation` i kombination med något av alternativen i enumerationen `ActivationOption`.

4.2.3 Registrera COM+-applikation

Vi kan om vi vill registrera våra .NET-komponenter i en COM+-applikation på samma sätt som vi gör i COM, d.v.s. med guiderna i Komponenttjänster. Eller så kan vi använda verktyget `REGSVCS.EXE`[25] och attribut genom att skriva ”`regsvcs assemblynamn.ext`” i kommandotolken.

Vi kan också låta .NET automatiskt registrera våra komponenter i COM+. Detta är endast ett alternativ som rekommenderas vid utveckling av komponenter.

Observera att vi måste ha administratörsrättigheter på dator för att detta ska fungera!

4.3 Transaktioner [UTVECKLA]

För transaktioner applicerar vi attributet `Transaction` på klassen med något av alternativen i enumerationen `TransactionOption`. För respektive metod använder vi sen metoden SetComplete. [HUR???]

Vi kan även applicera attributet `AutoComplete` på metoderna för att låta .NET hantera anropandet av SetComplete. Om vi använder detta attribut så måste dock några villkor vara uppfylld om vi ska använda vår komponent ifrån COM. Observera att fel som uppstår påverkar.

- [**ATT GÖRA:** Lista villkor som gäller för AutoComplete]
- [**ATT GÖRA:** Flytta delar av detta till tidigare avsnitt i kapitel… Gjort?]

4.4 Objektpoolning [UTVECKLA]

4.5 Klienter till komponenter som använder komponenttjänster

Observera att även klienter (skapade för .NET) till komponenter som använder komponenttjänster måste ha en referens till namnutrymmet `System.EnterpriseServices` – utan det så kompilerar inte klienten.

[25] Filen `REGSVCS.EXE` finns antagligen i mappen `%WINDIR%\Microsoft.NET\Framework\v1.0.3705\` (beroende på version av .NET Framework).

5 Exempel: En n-lager applikation [UTVECKLA]

I detta kapitel beskrivs hur min kära datorbokningsapplikation skulle kunna implementeras. Applikationen är den samma som den utvecklad i COM/COM+. Eftersom .NET är nytt så kommer jag förutsätta att vi använder ett jämngammalt operativsystem, d.v.s. Windows 2000[26] eller XP med COM+, för att utveckla applikationen. (Vilket förövrigt är det operativsystem som jag utvecklade exemplet i. ☺) Men exempel bör även fungera på Windows NT 4 med Option Pack installerat – dock med smärre justeringar (t.ex. att tillämpningar i COM+ heter paket i MTS ☺). Överallt där jag refererar till COM+ bör med andra ord kunna ersättas med MTS.

I denna första .NET-version av exemplet returnerar metoderna i affärslagret DataSet-objekt. En annan lösning vore att använda värdeobjekt (enkla klasser) som kan returneras i vektorer (av ”typen” Collection). På så sätt kan det grafiska gränssnittet utvecklas ”rent” objektorienterat. (Eventuellt kommer en ny version av denna applikation att utvecklas – då med värdeobjekt. Och eventuellt kommer denna applikation att ”portas” till C#. Men detta är framtida projekt…)

5.1 Komponenter

Eftersom vi inte längre ”behöver” göra ett datalager med komponenter så använder jag en klass `DBProxy` för att hantera kommunikationen med datakällan. Därmed behövs inte komponenterna (klasserna) Read och Write som i exemplet i VB6.

Affärslagret består av de två klasserna (komponenterna?) Läs och Skriv med samma (eller liknande) metoder som i VB6-exemplet i COM/COM+. Även om .NET är relativt nytt så kommer klassen Läs implementeras med namnet `Las` för att undvika eventuella problem med svenska tecken.

Nedan följer beskrivningar för att skapa projektet samt klasserna `DBProxy`, `Las` och `Skriv`.

5.1.1 Skapa och förbered projekt

Först måste vi skapa ett projekt s

1. Skapa ett nytt projekt av typen Class Library. Döp projektet till t.ex. `BPnDatorBokningVB` (ersätt gärna prefixet ”`BPn`” med er egen användaridentitet i nätverket).
2. Högerklicka på projektets gren References i Project Explorer och välj **Add Reference…** för att visa dialogrutan Add Reference.
3. Leta upp och markera **System.EnterpriseServices**. Klicka på knappen **Select** och sen **OK** för att stänga dialogrutan.

I varje klass som ska användas som komponent (samt projektets fil `AssemblyInfo.vb`) så lägger vi sen till nedanstående två Imports-satser så att vi slipper upprepa namnutrymmen hela tiden.

```
Imports System.EnterpriseServices
Imports System.Runtime.InteropServices
```

[26] Windows 2000 är äldre än .NET. Men det var den senaste serverversionen som fanns när .NET släpptes. Numera finns även Windows 2003 (Win 2K3) Server.

Nu är det dags att skapa en ”nyckelfil” (`.SNK`), om ni inte redan gjort det, och lägga till sökvägen till den.[27] (Se beskrivning i avsnittet *Skapa en delad assembly* ovan.) Vi namnger även tillämpningen i COM+ (eller paketet i MTS) samt vilken typ (server eller bibliotek) av tillämpning (eller paket) som vi ska använda.

4. Skapa en nyckelfil (t.ex. `bpn.snk` – ersätt gärna namnet ”`bpn`” med er egen användaridentitet).
5. Öppna projektets fil `AssemblyInfo.vb` och lägg till nedanstående rader efter attributet `AssemblyVersion` (ersätt sökvägen till nyckelfilen, om annan, samt namn och prefix ”`BPn`” med egna användaridentiteten i nätverket).

```
<Assembly: AssemblyKeyFile("C:\Student\bpn.snk")>           'Sökväg till nyckelfil
<Assembly: ApplicationName("BPnDatorBokningApp")>           'Namn på tillämpning
<Assembly: ApplicationActivation(ActivationOption.Server)>  'Typ av tillämpning
```

När projektet är förberett så kan vi börja skapa klasserna (komponenterna?) i projektet. Vi börjar med klassen DBProxy och testar att den fungerar. När vi vet att klassen fungerar så kan vi börja anpassa den för COM+ (eller MTS ☺).

5.1.2 Datakomponenten DBProxy

Klassen innehåller två metoder:

- `ExecuteQuery()` – utför den SELECT-sats som skickas som parameter till metod och returnerar resultatet i ett DataSet-objekt.
- `ExecuteUpdate()` – utför den INSERT-, UPDATE- eller DELETE-sats som skickas som parameter till metod och returnerar ett heltal med antal poster som påverkades av SQL-satsen.

Valet att skapa dessa metoder baseras på att en SELECT-sats hämtar en eller flera poster från datakälla medan övriga typer av SQL-satser uppdaterar data i datakälla. Att hämta data innebär att en hel del objekt måste skapas (t.ex. DataAdapter och DataSet), medan att uppdatera kan ske med enbart Connection- och Command-objekt.

Döp om klassen `Class1` till `DBProxy` samt filen från `Class1.vb` till `DBProxy.vb`.

```
Public Class DBProxy
  Private Const cstrConn As String = "Provider=Microsoft.Jet.OLEDB.4.0;" _
          & "Data Source=C:\Student\Datorbokning.mdb;Persist Security Info=False"

  'Funktion som utför en SELECT-sats och returnerar poster i ett DataSet-
  ' objekt. Om inga poster returneras Nothing
  Public Function ExecuteQuery(ByVal Sql As String) As DataSet
    Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
    Dim adoDA As OleDb.OleDbDataAdapter, adoDS As DataSet

    adoConn = New OleDb.OleDbConnection(cstrConn)
    adoConn.Open()
    adoCmd = adoConn.CreateCommand()
    adoCmd.CommandText = Sql
    adoCmd.CommandType = CommandType.Text
    adoDA = New OleDb.OleDbDataAdapter(adoCmd)
    adoDS = New DataSet()
    adoDA.Fill(adoDS)
    adoConn.Close()
```

[27] Man behöver endast skapa en nyckelfil, d.v.s. inte en för varje projekt eller lösning.

```
      Return adoDS
    End Function

    'Funktion som utför en INSERT-, UPDATE- eller DELETE-sats och returnerar
    ' antalet poster som påverkades av SQL-satsen.
    Public Function ExecuteUpdate(ByVal Sql As String) As Integer
      Dim adoConn As OleDb.OleDbConnection, adoCmd As OleDb.OleDbCommand
      Dim intAntal As Long

      adoConn = New OleDb.OleDbConnection(cstrConn)
      adoConn.Open()
      adoCmd = adoConn.CreateCommand()
      adoCmd.CommandText = Sql
      adoCmd.CommandType = CommandType.Text
      intAntal = adoCmd.ExecuteNonQuery()
      adoConn.Close()
      Return intAntal
    End Function

  End Class
```

5.1.3 Affärskomponenten Las

Metoderna i denna klass bygger på att vi skapar en SQL-sats som vi sen skickar som parameter till metoderna i en instans av klassen DBProxy (som vi skapade ovan).

1. Högerklicka på projektets namn (BPnDatorBokningVB), välj **Add** från menyn som visas och sen **Add Class…** från undermenyn.
2. Ändra namnet på filen till `Las.vb`. Därmed kommer även klassen i filen att få namnet `Las`.
3. Fyll i koden som saknas i exempel nedan.

```
  Imports System.EnterpriseServices
  Imports System.Runtime.InteropServices

  <Transaction(TransactionOption.Supported)> _
  Public Class Las
      Inherits ServicedComponent

      'Metod som returnerar alla salsID
      Public Function GetSalar() As DataSet
          Dim strSql As String, objProxy As DBProxy, adoDS As DataSet

          strSql = "SELECT * FROM tblSalar"

          'Skapa proxy-objekt och utför fråga
          objProxy = New DBProxy()
          adoDS = objProxy.ExecuteQuery(strSql)

          ContextUtil.SetComplete()        'Meddela COM+ att metod är klar

          Return adoDS                     'Returnera DataSet
      End Function

      'Metod som returnerar alla datorer i en sal
      Public Function GetDatorerForSal(ByVal Sal As String) As DataSet
          Dim strSql As String, objProxy As DBProxy, adoDS As DataSet

          strSql = "SELECT * FROM tblDatorer WHERE SId='" & Sal & "'"

          'Skapa proxy-objekt och utför fråga
          objProxy = New DBProxy()
          adoDS = objProxy.ExecuteQuery(strSql)

          ContextUtil.SetComplete()        'Meddela COM+ att metod är klar

          Return adoDS                     'Returnera DataSet
      End Function
```

```
    'Metod som returnerar alla bokningar för en sal
    Public Function GetBokningarForSal(ByVal Sal As String) As DataSet
        Dim strSql As String, objProxy As DBProxy, adoDS As DataSet

        strSql = "SELECT * FROM tblBokningar WHERE SId='" & Sal & "'"

        'Skapa proxy-objekt och utför fråga
        objProxy = New DBProxy()
        adoDS = objProxy.ExecuteQuery(strSql)


        ContextUtil.SetComplete()        'Meddela COM+ att metod är klar

        Return adoDS                     'Returnera DataSet
    End Function

End Class
```

5.1.4 Affärskomponenten Skriv

Även metoderna i denna klass bygger på att vi skapar en SQL-sats som vi sen skickar som parameter till metoderna i en instans av klassen DBProxy (som vi skapade ovan).

1. Högerklicka på projektets namn (BPnDatorBokningVB), välj **Add** från menyn som visas och sen **Add Class…** från undermenyn.
2. Ändra namnet på filen till `Las.vb`. Därmed kommer även klassen i filen att få namnet `Las`.
3. Fyll i koden som saknas i exempel nedan.

```
Imports System.EnterpriseServices
Imports System.Runtime.InteropServices

<Transaction(TransactionOption.Supported)> _
Public Class Skriv

    'Metod som skriver en bokning till datakälla (om inte redan bokad)
    Public Function Boka(ByVal Sal As String, ByVal Dator As Byte, _
    ByVal Dag As DateTime, ByVal Namn As String) As String
        Dim objProxy As DBProxy, strSql As String

        strSql = "INSERT INTO tblBokningar(SId, DId, BDatum, Namn) VALUES('" _
            & Sal & "', " & Dator & ", #" & Dag & "#, '" & Namn & "')"

        objProxy = New DBProxy()

        Try
            objProxy.ExecuteUpdate(strSql)

            ContextUtil.SetComplete()
        Catch ex As Exception

            ContextUtil.SetAbort()
        End Try
    End Function
End Class
```

5.2 Formulär (WinForms)

5.3 Formulär i ASP.NET (WebbForms)

5.4 En alternativ lösning med värdeobjekt

I lösningen ovan returneras data i DataSet-objekt. En fördel med denna lösning är att DataSet-objekt kan bindas till datakontroller[28]. Genom att binda DataSet till datakontroller så slipper vi skriva en massa kod för att presentera data i DataSet. Detta är mycket användbart i t.ex. ASP.NET. En nackdel är att klienten måste hantera DataSet.[29]

En annan lösning är att använda ”värdeobjekt”. Ett värdeobjekt är (egentligen) en klass som främst innehåller attribut (eller egenskaper, instansvariabler med motsvarande accessmetoder). Genom att skapa dessa enklas klasser så kan de används för att kommunicera data mellan klienter och komponenter. Dessa klasser intstalleras **inte** i MTS/COM+.

En ”fördel” med värdeobjekt är att de oftast motsvara klasserna i de ursprungliga klassdiagrammen, d.v.s. de som vi tog fram i analysfasen för applikationen.[30] Ytterligare en fördel är att klienten kan lösas ”rent objektorienterat”.

Värdeobjekten placeras i ett eget paket (Class Library, d.v.s. DLL) som måste finnas tillgängligt på både server och klient.

6 ASP.NET [UTVECKLA]

Precis som vi kan använda Active Server Pages (ASP) i COM/COM+ så kan vi använda ASP.NET i .NET. Skillnaden är bl.a. att ASP.NET numera är kompilerad kod i motsats till ASP:s tolkade skriptkod.

Annat som skiljer ASP.NET från ASP är att filändelsen är .ASPX (istället för .ASP).

[28] Med datakontroller menar jag t.ex. tabeller och inte de datakontroller som fanns i VB6.
[29] Personligen tycker jag det är lättare att jobba med objekt då dess struktur kan utläsas från dess publika egenskaper och metoder.
[30] Jag kallar dem ursprungliga då vi måste anpassa klasser för att installeras i MTS/COM+, d.v.s. attribut tas bort då komponenter i MTS/COM+ inte får ha tillstånd.

7 Tips och trick

I detta kapitel beskrivs hur man kan underlätta utvecklingen av applikationer i .NET.

8 Litteratur

Denna beskrivning har utgått från sammanfattningarna *Visual Basic 6 för komponenter* och *Komponenter med DCOM/MTS*. Nedan finns annan litteratur som använts som referensmaterial.

- Anderson, R. et al, *Professional ASP.NET*, Wrox Press, 2001.
- Barwell, F. et al, *Professional VB.NET*, Wrox Press, 2001.
- Hoffman, K., et al, *Professional .NET Framework*, Wrox Press, 2001.
- Homer, A. & D. Sussman, *ASP.NET Distributed Data Applications*, Wrox Press, 2002.
- Löwy, J., *COM and .NET Component Services*, O'Reilly, 2001.
- MacClure, W.B. & J.J. Croft IV, *Building Highly Scalable Database Applications with .NET*, Wiley Publishing, 2002.
- Robinson, Ed, et al, *Upgrading MS Visual Basic 6.0 to MS Visual Basic.NET*, Microsoft Press, 2002.
- Robinson, S, et al, *Professional C#*, Wrox Press, 2001.
- Roman, S., et al, *VB.NET Language in a Nutshell*, O'Reilly, 2001.
- Thai, T. & H.Q. Lam, *.NET Framework Essentials*, O'Reilly, 2002.

samt webbsidor hos Microsoft (bl.a. MSDN) och webbplatsen GotDotNet.com.