MÄLARDALENS HÖGSKOLA
Institutionen för ekonomi och informatik

Komponenter med COM (och COM+)

EI0230

Komponentbaserad applikationsutveckling

oktober 2003

Om denna sammanfattning

Denna sammanfattning avser att ge en inblick i komponentteknologier med Microsofts COM (och COM+) och utgick ursprungligen från Kirtlands bok *Designing Component-Based Applications*. Namnet på sammanfattningen syftar till att visa på att COM är grunden till COM+, d.v.s. COM+ är en utökning av COM och att om vi förstår COM så har vi börjat förstår COM+ (COM+ omfattar mer än COM).

Observera att detta **inte är en ersättning för eventuell kurslitteratur**. Sammanfattningen är inte uttömmande i ämnet, i programmeringsspråken som används i sammanfattning eller Windows-programmering. Innehållet i sammanfattning bygger på programmering i Visual Basic 6.0 och Visual C++ 6.0, men principerna bör kunna överföras till andra COM-kompatibla programmeringsspråk (t.ex. Borland Delphi).

Engelska ord utan en bra översättning har skrivits med *kursiv stil*. Kod har skrivits med typsnittet `Courier New` och längre kodstycken har inneslutits i ramar enligt följande:

```
Dim objKomp As MinServer.MinKomponent     'Kommentarer skrivs med fet stil
```

Förkortningen COM kommer att användas för både COM och det nyare COM+ när principen för båda är de samma. Då något bara avser COM eller MTS kommer förkortningarna COM/MTS användas eller, då bara COM+ avses, förkortningen COM+. (Förkortningen COM+ syftar till att mena det som inte är COM i COM+, d.v.s. främst ”nya” MTS.) Termen komponenter avser både klasser för och objekt (instanser) av komponenter om inte det uttryckligen anges.

Programvara från Microsoft som använts är *Visual Studio 6 Professional* (Service Pack 4).

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

Eskilstuna, oktober 2003

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

Innehållsförteckning

1 Microsofts Windows Distributed interNet Applications

Microsofts vision av distribuerade applikationer kallas ***Windows Distributed interNet Applications*** (DNA) och avser distribuerade applikationer i främst Windows miljö. DNA bygger på att applikationer utvecklas som mindre komponenter som sedan sammanfogas för att skapa en helhet. En komponent kan bytas ut mot en annan utan att hela applikationen påverkas, d.v.s. behöver kompileras om (förutsätter att gränssnittet för komponenten är oförändrat).

Microsoft har i DNA lyckats dölja de detaljer som tidigare gjort programmering av distribuerad applikationer tidskrävande – i Visual Basic nästan helt. Detta görs med olika teknologier som samarbetar för att skapa en miljö som gör det enklare att utveckla de distribuerade applikationerna. Dessa teknologier ska vi titta på lite längre fram.

1.1 Bakgrund till komponentteknologier

De mesta av programmering i Windows-miljö utgår ifrån att en exekverbar fil (EXE-fil) skapas för att köra programmen. För att EXE-filerna inte ska bli för stora så använder man sig av s.k. *dynamic link library*-filer (DLL-filer). DLL-filer innehåller kod (funktioner) som kan delas av flera program samtidigt. Och eftersom många program använder samma funktioner, t.ex. dialogrutor för att öppna och spara filer, så sparar man även plats i arbetsminnet (RAM) samt på hårddisken. Som begreppet DLL visar så laddas inte koden i DLL-filen förrän den behöver exekveras, s.k. dynamisk länkning, och koden anropas via ett s.k. *application programming interface* (API)[1]. DLL-filernas sökväg (i filsystemet) hårdkodas oftast in i EXE-filen.

Nackdelen med ”vanliga” DLL-filer är att om DLL-filen ändras, och därmed kompileras om, så kan den binära representationen av DLL-filen ändras vilket leder till att även applikationer som använder sig av DLL-filen måste kompileras om. Administration av uppdateringar kan bli tidskrävande.

Komponentteknologier löser detta genom att anropa kod genom gränssnitt (inte API) som är ”fristående” från implementationen av komponenten. D.v.s. om komponentens implementation ändras internt (kompileras om) så behöver inte gränssnittet påverkas. Applikationerna som använder sig av komponenten bör m.a.o. inte påverkas av detta. En ändring av antalet metoder eller metoders parametrar påverkar dock gränssnittet för komponenten och kräver att även applikationerna behöver kompileras om.

1.2 Komponenter

Begreppet komponent har många betydelser och litteratur om COM använder begreppet för att mena olika saker. I denna sammanfattning kommer vi använda begreppet komponent för att avse mjukvara som kan ersättas med annan mjukvara med samma gränssnitt (återigen inte API). I kod (programspråk) kan komponenter uttryckas som COM-servrar, -klasser och -gränssnitt samt som COM-objekt då koden exekverar. COM-gränssnitt behandlas i nästa kapitel.

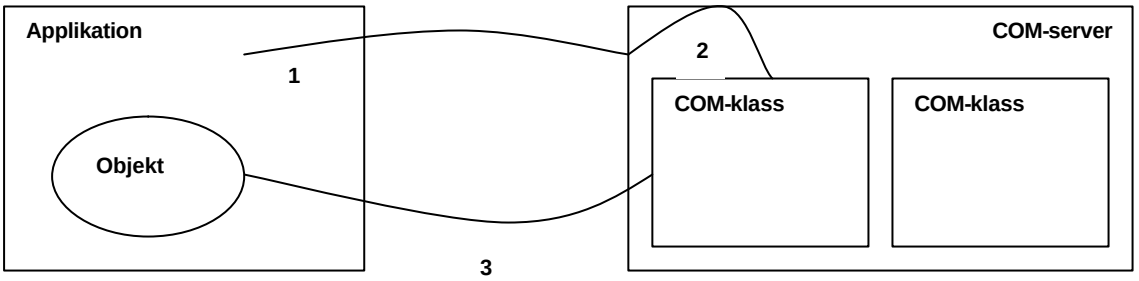
1.2.1 Klasser, objekt och servrar

En COM-klass kan sägas motsvara en klass i objektorienterade språk (OO-språk). Precis som klasser i OO-språk (vilka man kan använda för att implementera COM-klasser) så har COM-

[1] API-gränssnitt ska inte förväxlas med gränssnitt som COM använder! Gränssnitt är ett av de ord inom datorvärlden som har många betydelser, t.ex. så har användargränssnitt inget direkt med API eller COM att göra.

klasser egenskaper, i form av variabler (attribut), för att lagra tillstånd och metoder för att manipulera dessa variabler. Man använder dessa COM-klasser som mallar (eller fabriker) för att skapa COM-objekt (instanser).

Klasser samlas i COM-servrar (t.ex. i en DLL-fil) och serverns uppgift är att se till att objekt skapas samt att en pekare till objektet skickas tillbaka till applikationen som begärde COM-objektet. COM-servrar kan sägas motsvara projekt i utvecklingsmiljöer som Visual Basic och Visual C++.



Figur 1 - Applikation ber COM-server att hitta COM-klassen (1) som sen skapar ett COM-objekt (2). COM-klassen skapar objektet och skickar tillbaka en pekare till det nyskapade objektet (3).

1.2.2 Olika typer av komponenter

Komponenter enligt COM kan vara lokala komponenter eller serverkomponenter (distribuerade komponenter). **Lokala komponenter** används oftast på samma dator som applikationen medan **serverkomponenter** installeras i MTS (se *Teknologier i DNA*) – antingen på samma dator eller annan dator än klient.

Exemplen i denna sammanfattning är lokala komponenter medan exemplen i övriga sammanfattningar är serverkomponenter.

1.2.3 *Object Linking and Embedding* och ActiveX

Microsofts namngivning av sin komponentteknologi har förvirrat många under årens lopp. I begynnelsen kallades komponentteknologin för *Object Linking and Embedding* (OLE) vilket ersattes med namnet ActiveX för att numera enbart heta *Component Object Model* (COM).

Idag används termen OLE för teknologin som används för att bl.a. bygga sammansatta dokument och OLE bygger på COM. Termen ActiveX har återgått till att användas för det den användes för tidigare – visuella komponenter och komponenter som används över Internet. Än idag lever dock termen ActiveX kvar i t.ex. Visual Basic då man skapar en ActiveX DLL eller ActiveX EXE då man vill göra en komponent.

1.3 3-lagers arkitekturen

Den logiska strukturen för Windows DNA bygger på tre lager (*tiers*): **presentation**, **affärslogik** och **dataåtkomst**. Dessa tre lager samarbetar för att skapa en helhet – en applikation. Beroende på applikationens storlek/komplexitet så varierar storleken på, och gränserna mellan, lagren. Det finns alltså inga strikta regler som säger att en funktion i applikationen ska implementeras i ett visst lager.

1.3.1 Presentations lager

Presentations lager används för att kommunicera med användaren, d.v.s. vara ett gränssnitt mot användaren, och dess uppgift bör inte vara mycket mer än så. Detta lager implementeras oftast som ett vanligt exekverbart program (.EXE) eller med hjälp av en webbserver och ett skriptspråk (t.ex. VBScript i *Active Server Pages*, ASP). I det första fallet placeras EXE-filen på klientdatorn och i andra fallet används en webbläsare på klienten.

1.3.2 Affärslogik lager

I detta lager, mellanlagret, sker det mesta av exekveringen. Lagrets namn, affärslogik, kommer sig av att här sker kontroller så att organisationens affärsregler efterföljs. Exempel på en affärsregel är att ingen försäljning över 1000 kronor får ske utan att kreditupplysning har gjorts på kunden. Genom att centralisera affärslogiken till ett ställe så blir uppdatering av applikationen lättare. Om transaktioner är aktuella, t.ex. om uppdatering av flera datakällor sker, så är det mellanlagrets uppgift att hantera transaktionerna.

Lagret implementeras som komponenter som kan exekvera antingen på en server eller på klienten. I vissa fall, t.ex. när en webbserver används, så kan en del av applikationens gränssnitt skapas/genereras i detta lager.

Enligt en del författare (Lhotka [98]) så bör man dela upp detta lager i ett dellager som är fokuserat på användargränssnitt och ett dellager som är fokuserat på datalagring. Det första dellagret kan då placeras på samma dators som presentationslagret för att ge ett rikare gränssnitt (mer interaktivt). Datafokuserade dellagret placeras på en applikationsserver.

1.3.3 Dataåtkomst lager

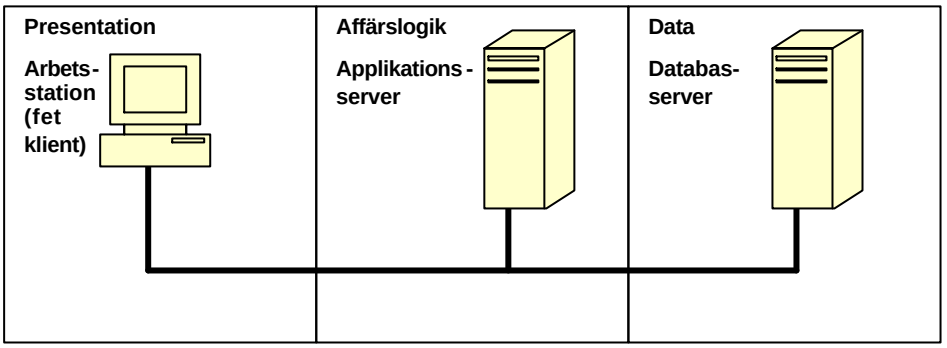
Det sista lagret, dataåtkomst, har endast till uppgift att hantera kommunikation med datakällor (*data providers*) och förse lagret för affärslogik med data. Datakällor behöver inte bara vara traditionella relationsdatabaserna utan kan vara flata (text-)filer, katalogtjänster, e-postsystem, m.m..

Lagret implementeras som komponenter som oftast exekvera på servern med datakällan.

1.3.4 Lagrens fysiska placering

Komponenter i alla tre lager kan placeras på en och samma dator eller delas upp på tre typer av datorer: klientdator, applikationsserver och databasserver. Var komponenterna placeras beror på antalet samtidiga användare och organisationens resurser (hur mycket pengar man vill spendera på servrar).

Om man använder lokala komponenter så smälts ofta alla tre lager ihop och placeras på klienten. Använder man istället serverkomponenter så installerar man oftast komponenterna i MTS som finns på separata serverar – antingen en kombinerad applikations- och databasserver eller två dedicerade servrar.



1.4 Teknologier i DNA

I Windows NT 4 används ***Component Object Model*** (COM), och eventuellt även ***Microsoft Transaction Server*** (MTS), för att skapa en miljö för komponenter. För distribuerade komponenter, d.v.s. komponenter på två eller fler datorer, används ***Distributed COM*** (DCOM) för kommunikation mellan komponenter på olika datorer. COM och MTS har slagits samman till **COM+** i Windows 2000 men fungerar på ett liknande sätt. COM+ är bakåt kompatibelt och komponenter som skrivit för COM/MTS fungerar även under COM+.

För dataåtkomst används ***Microsoft Data Access Components*** (MDAC) som bygger på principen ***Universal Data Access*** (UDA), d.v.s. att alla typer av datakällor ska kunna användas med ett enhetligt gränssnitt. Som ett sätt att skapa gränssnitt kan man använda Microsofts ***Internet Information Server*** (IIS).

1.4.1 Component Object Model

Component Object Model (COM) är både en standard och en implementation. D.v.s. COM specificerar hur kommunikation med komponenter sker och implementationen av COM ger tjänster som att hitta och skapa instanser (objekt) av komponenter.

COM är en **språkoberoende** (*language independent*) teknologi, d.v.s. komponenter kan skrivas med vilket som helst språk (som förstår COM, d.v.s. är COM-kompatibelt). Komponenterna kan sen anropas av ett annat språk (också det COM-kompatibelt). T.ex. så kan en komponent skrivas i C++ för att skapa effektiv binärkod och anropas av Visual Basic eller Delphi som är bättre för grafiska gränssnitt mot användaren.

COM är också en **platsoberoende** (*location independent*) teknologi, d.v.s. på vilken dator komponenten exekverar ska inte spela någon roll. Programmet som använder sig av komponenten kan t.ex. var en arbetsstation (med Windows 98) och komponenten kan vara placerad på samma dator eller en server (med Windows NT/2000) – anropet av en komponents metod är det samma ur programmerarens synvinkel. Givetvis kommer svarstiderna bli längre om komponenten är placerad på en annan dator och måste nås över nätverket.

Språkoberoendet och platsoberoendet löses genom att separera komponentens gränssnitt från implementationen av komponenten. Komponenters gränssnitt behandlas i nästa kapitel. Platsoberoendet skapas av bl.a. DCOM.

1.4.2 Distributed COM

Distributed COM (DCOM) är en förlängning av COM som gör att programmerare av en distribuerad applikation inte behöver bry sig om var en komponent exekverar. Anropet av

komponenters metoder sker på samma sätt, oavsett om komponenterna finns lokalt eller på en annan dator. DCOM hanterar kommunikation mellan datorer och gör detta transparent.

Det behöver inte skrivas någon speciell kod (av programmerare) för att göra en komponent distribuerad. En sökväg i systemregistret[2] (*system registry*), och eventuellt en minimal bit av binärkod (*proxy*), skapas för distribution till klienterna som ska använda komponenten. Binärkoden skapas med hjälp av COM/MTS-programvara via ett grafiskt gränssnitt. Mer om COM och DCOM finns i nästa kapitel.

1.4.3 Microsoft Transaction Server

Microsoft Transaction Server (MTS) används för skapande av instanser av komponenter och hanterar, om man vill, även transaktioner. Namnet MTS är missvisande då MTS inte bara hanterar transaktioner. MTS används även som surrogatprocess för att exekvera komponenters kod i DLL-filer på annan dator.[3] Man kan säga att det är löst nu då MTS ingår i COM+ och är alltså inte en separat programvara längre. MTS behandlas i sammanfattningen *Komponenter med DCOM/MTS*.

1.4.4 Microsoft Data Access Components

Microsoft Data Access Components (MDAC) är inte direkt en del av DNA, men är i stort sett en förutsättning för att skapa distribuerade applikationer som använder sig av databaser. MDAC består bl.a. av *ActiveX Data Objects* (ADO) och OLE DB, som faktiskt är komponenter som använder sig av COM. (Mer om MDAC i Visual Basic finns i sammanfattningen *Visual Basic 6 för komponenter*.)

1.4.5 Microsoft Internet Information Server

Microsofts *Internet Information Server* (IIS) har utvecklats och nått version 5 i samband med att Windows 2000 släpptes. IIS kan användas som ett webbgränssnitt mot distribuerade applikationer med COM. Komponenterna kan exekveras m.h.a. skriptkod som exekvera på webb-servern genom *Active Server Pages* (ASP) och till användarens webbläsare skickas vanlig HTML-kod. Om användaren använder Microsoft Internet Explorer (MSIE) kan man öka funktionaliteten i webbapplikationen genom att t.ex. skicka lösgjorda tabeller (*disconnected recordsets*) som kan manipuleras direkt i MSIE.

[2] I systemregistret, vanligtvis bara kallat registret, lagras inställningar för program, registreringar av komponenter, m.m.. Registret ersätt INI-filerna i Windows 3.x under Windows 9x/NT/2000. OBS! Registret i de olika versionerna av Windows har olika uppbyggnad!
[3] En DLL-fil kan inte exekvera på egen hand utan måste anropas från exekverande kod, t.ex. en EXE-fil.

(Denna sida har avsiktligt lämnats blank.)

2 COM:s funktion och detaljer som påverkar exekvering av komponenter

Komponenter och dess delar identifieras av ett 128-bitars nummer kallat GUID (*Globally Unique Identifier*). Dessa GUID är dock inte vidare praktiska att använda vid programmering, utan programmerare använder oftast komponenters namn. En av komponenternas delar som identifieras av ett GUID är gränssnitt, genom vilka all kommunikation med komponenten sker. Ytterligare en del i komponenter med GUID är klasserna som implementerar gränssnittet, och utifrån vilka instanser av komponenterna skapas.

Det finns ett antal inställningar (faktorer) som påverkar exekveringen av komponenter. Några av dessa inställningar kan anges vid skapande av komponenter och andra vid installation av COM-applikationer i MTS/COM+ (i vilka komponenter ingår). Några av dessa inställningar kan anges både vid skapande och installation i COM-applikationer.

I Visual Basic döljs många av dessa inställningar (detaljer) och andra inställningar kan endast anta vissa värden. En insikt i ämnet är bra att ha för att förstå implikationerna av dessa värden, t.ex. att C++ är lämpligare än Visual Basic för vissa komponenter, samt att C++ inte döljer riktigt lika mycket (även om *Active Template Library*, ATL, hjälper till med detta).

Saker som påverkar exekveringen:

- hur synkronisering av komponenters exekverande sker (trådmodeller).
- var komponenter exekverar (in-process, out-of-process).

2.1 Globally Unique Identifiers

Eftersom det inte (enligt Microsoft) räcker med namn (strängar) för att unikt identifiera komponenter (och annat i Windows-miljön) används ett 128-bitars nummer kallat GUID för detta. Dessa GUID skrivs med hexadecimala tal på formen `{D541DF1C-0E2C-4c21-B132-3B7D38ED777A}` och genereras ofta automatiskt av programmeringsmiljön (med hjälp av datorns nätverkskort samt aktuell tid och datum[4]).[5] Man kan kalla dessa GUID för fysiska namn.

Givetvis är inte ett GUID vidare tilltalande för oss människor att använda, och därför används ett logiskt namn (*programatic ID*, ProgID) när programmeraren skriver kod. I Visual Basic översätts de logiska namnen till fysiska namn automatiskt och i Visual C++ finns en funktion som kan användas. Det finns ett fåtal tillfällen då man tvingas skriva dessa GUID – främst i hemsidor och skriptkod i hemsidor.

I COM används registret för att lagra kopplingarna mellan logiska och fysiska namn samt omvänt.

2.2 Gränssnitt

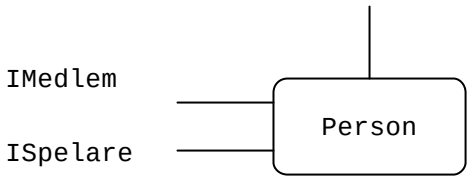
All kommunikation med komponenter sker genom komponenters gränssnitt (*interface*). Definitionen för ett gränssnitt är ”en samling av semantiskt relaterade metoder”, d.v.s. en definition av metoderna och dess parametrar. Alla gränssnitt börjar enligt konventioner med stora bokstaven ”I”, t.ex. `IMedlem` och `ISpelare` (ej helt sant i Visual Basic).

[4] Saknar datorn nätverkskort kan bara GUID:et garanteras vara unikt på den datorn.
[5] Intresseklubben: Ett GUID är 32 hexadecimala tecken långt (bortsett från bindestreck) vilket kommer sig av att 4 bitar (binära siffror) motsvarar ett hexadecimalt tal (128/4 = 32). 4 bitar kan m.a.o. anta 16 kombinationer (med 0 inräknat!).

2.2.1 Introduktion till gränssnitt

Komponenter enligt COM kan ha flera gränssnitt. T.ex. så kan en person i en fotbollsklubb både vara medlem i klubben och samtidigt vara en aktiv spelare. Som medlem har personen t.ex. namn och adress medan som spelare har personen t.ex. namn och position på planen. Detta skulle kunna implementeras som komponenten `Person` som har gränssnitten `IMedlem` och `ISpelare`. I bilden till höger visas komponenten `Person` som en rektangel (med rundade hörn) och gränssnitten stickande ut som namngivna ”klubbor” (*lollypops* – därav namnet *lollypop diagram*). Den tredje icke namngivna klubban (som sticker rakt upp) representerar gränssnittet `IUnknown` (se mer nedan).

Det GUID som används för att unikt identifiera ett gränssnitt kallas för *Interface Identifier* (IID). Observera att detta GUID endast används för att identifiera gränssnittet, inte komponentens implementation (koden). Detta sker med ett GUID kallat *Class Identifier* (CLSID).



Gränssnitt som skapats av Microsoft brukar kallas standard gränssnitt och gränssnitt skapade av programmerare brukar kallas **anpassade gränssnitt** (*custom interfaces*).

2.2.2 Gränssnitt som kontrakt

Gränssnitt representerar ett kontrakt mellan anropade program och komponenter. Detta leder till att ”publicerade” komponenters gränssnitt inte bör ändras, d.v.s. antalet metoder och metodernas signatur. Detta förhindrar dock inte att implementationen av komponenten (koden) inte kan ändras, t.ex. för att korrigera buggar i koden. Om en DLL-fil i traditionell programmering skulle korrigeras, och därmed kompileras om, så behöver även klientprogram som använder sig av DLL-filen kompileras om. Med komponenter så behöver inte klientprogram kompileras om (om inte gränssnittet ändras). Givetvis kommer det tillfällen då en komponent kan behöver utökas med ny funktionalitet och därmed fler metoder. Detta kan lösas genom att behålla det gamla gränssnittet och ge komponenten ett nytt gränssnitt för den nya funktionaliteten, d.v.s. en ny version av gränssnittet (*interface versioning*).

Liksom klasser i objektorienterade språk så kan gränssnitt ärva från andra gränssnitt och alla gränssnitt ärver ursprungligen från gränssnittet `IUnknown`.

2.2.3 IDL-filer

IDL[6]-filer används för att skapa gränssnitt för komponenter i COM och IDL-filerna kompileras till *type libraries* (TLB-filer). Ett exempel på hur en IDL-fil i COM skulle kunna se ut ses nedan. Gränssnittet heter `_Slump` och har en metod med namnet `Slumpa()` (se kapitlet *En första komponent i Visual Basic*).

```
[
  uuid(4E266A9A-A575-45E6-9EE8-FB3318ED1FB9),
  version(1.0),
  hidden,
  dual,
  nonextensible
]
dispinterface _Slump {
    properties:
    methods:
        [id(0x60030000)]
        VARIANT Slumpa([in, out] short* Max);
```

[6] *Interface Definition Language*.

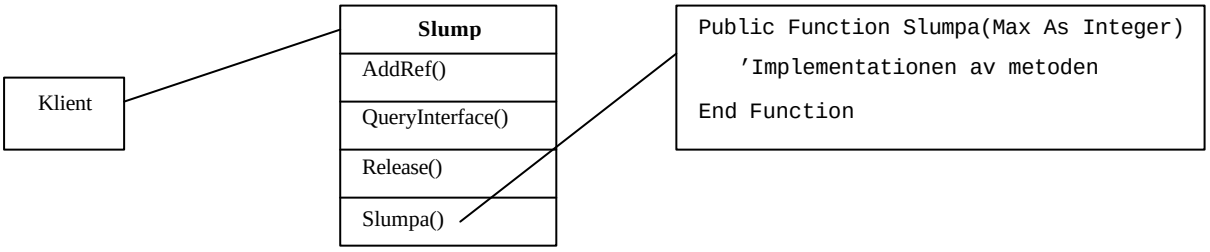
```
};
```

Ett undantag från detta är bl.a. Visual Basic som döljer detaljer om gränssnitt genom att automatiskt generera dessa *type libraries,* vilket ingår i EXE- eller DLL-filen för komponenten. Gränssnitten i Visual Basic motsvarar klassen och dess metoder som implementerar komponenten. D.v.s. en publik Visual Basic-klass med dess publika egenskaper och metoder blir gränssnittet för en komponent (se exempel i senare kapitel). Namnet på (standard) gränssnittet för komponenten är det samma som klassen, men föregås av ett understyckningstecken (”_”), t.ex. `_Slump`.

2.2.4 vtable

När en komponent kompileras så skapas en ”tabell”, kallad *Virtual Method Table* (*vtable* eller *vtbl*), med alla metoder för komponenten. När ett objekt skapas returneras en pekare (egentligen en pekare till en pekare) till komponentens gränssnitt, d.v.s. komponentens *vtable*. Och alla anrop till komponentens metoder sker via komponentens *vtable*.

En *vtable* innehåller inte själva implementationen (koden) för metoder, utan pekare till koden.



Figur 2 - Klienten har en pekare (referens) till komponentens vtable som i sin tur pekar på implementationen av metoderna.

I figuren ovan visas fyra metoder – metoden `Slumpa()` och tre metoder till. Dessa tre extra metoder ärvs från gränssnittet `IUnknown` (se nedan) och finns alltid med i komponenters *vtable*.

En komponent har en *vtable* för varje gränssnitt som komponenten stödjer. Denna konstruktion med *vtables* används för att bl.a skapa språkoberoendet för komponenter i COM.

2.2.5 IUnknown

Gränssnittet `IUnknown` finns för att ge komponenter några grundläggande egenskaper och består av 3 metoder: `QueryInterface()`, `AddRef()` och `Release()`. Metoden `QueryInterface()` används för att fråga efter en pekare till ett visst gränssnitt och som parameter till metoden sänds ett IID. De andra två metoderna används för att hantera komponenters existens. När ett objekt skapas, eller en referens/pekare sätts till att peka på objektet, så anropas metoden `AddRef()` för att öka på en räknare som håller reda på antal referenser till objektet. Och när anropande program inte längre behöver objektet anropas `Release()` för att minska räknaren. När räknaren med referenser når noll så kan komponenten förstöra sig.

I Visual Basic anropas `AddRef()` automatiskt då New-operatorn används eller då variabel sätts att refe rera till ett objekt med Set-operatorn. Likaså så anropas `Release()` automatiskt då referensen sätts till `Nothing` eller då variabeln inte längre är giltig (*out of scope*), t.ex. då metoden som variabeln deklarerades i har avslutats.

2.2.6 IDispatch

Kompilerade program (t.ex. VB och C++) använder något som kallas tidig bindning (*early-binding*). Detta innebär att metodanrop och dess parametrar kontrolleras så att de är riktiga vid kompilering. Därmed minskas antalet fel vid exekvering (*run-time errors*).

Skriptspråk, som *Visual Basic for Applications* (VBA) och VBScript i ASP, tolkas rad för rad vid exekvering och använder därför sen bindning (*late-binding*) till komponenter, d.v.s. kontrollen om anropet är riktigt sker först vi exekvering. Ett felstavat namn på en metod kan leda till ett fel vid exekvering.

Skriptspråk är ofta generella och kan inte känna till alla gränssnitt, utan de behöver ett annat sätt att anropa komponenters metoder (genom sen bindning). Dessa anrop sker via komponenters gränssnitt `IDispatch` och brukar kallas för *Automation*. Detta leder till att komponenter som ska användas i skriptspråk måste ärva även gränssnitt `IDispatch`.

Att använda gränssnittet `IDispatch` för att skapa och anropa metoder i komponenter kan inte ske direkt som med komponenters *vtable*. Först måste namnet (`metodX()`) på metoden översättas till ett nummer, kallat *Dispatch ID* (DispID)[7], i komponentens *vtable*. För det använder man sig av metoden `GetIDsOfNames()` (i gränssnittet `IDispatch`) som returnerar DispID för metoden (`metodX()`). Sen anropar man `Invoke()`, i gränssnittet `IDispatch`, och skickar som parametrar DispID för metoden (`metodX()`) man vill anropa i komponenten samt parametrarna till metoden. D.v.s. det krävs två anrop för att anropa en metod (`metodX()`) i en komponent – `GetIDsOfNames()` och `Invoke()`. Att använda sen bindning är m.a.o. inte det mest effektiva sättet att exekvera kod.

Parametrar som skickas via gränssnittet `IDispatch` måste vara av typen Variant. Språk som C++, i motsats till VB, som inte direkt stödjer datatypen Variant måste konvertera parametrarna innan anrop till `Invoke()`.

I t.ex. Visual Basic så ärvs gränssnittet `IDispatch` automatiskt och komponenter kan användas direkt av skriptspråk. I C++ måste specificera att man vill stödja gränssnittet.

2.2.6.1 Dual interfaces

Även om anrop av komponenters metoder via *vtables* är mer effektivt än att anropa metoderna via gränssnittet `IDispatch`, så vill man ibland att komponenter ska kunna användas av både skriptspråk, via `IDispatch`, och via vtable. Om så är fallet så implementerar man dubbla gränssnitt (*dual interfaces*).

En komponent skapad i Visual Basic stödjer som standard både gränssnittet `IDispatch` och *vtables*. Med C++ kan man t.ex. välja om man vill stödja båda när man använder ATL-guiderna.

2.3 Klasser och objekt

COM-klasser implementerar ett eller flera gränssnitt och utifrån dessa klasser skapas sen COM-objekt som används för att anropa metoder i gränssnittet. Efter att en klass har skapat ett objekt behövs inte en referens/pekare till klassen (annat än att den innehåller koden för metoderna – vilket dock hanteras av objektet själv). D.v.s. klassen används endast för att skapa objektet – all kommunikation med objektet sker sedan via objektets gränssnitt.

[7] Ett DispID är inte ett GUID utan endast ett ordningstal för metodens relativa position till andra metoder i gränssnittet.

Klasser identifieras, som nämnt tidigare, av ett fysiskt namn kallat ClsID (Class ID), vilket är ett GUID. Men för att underlätta för programmeraren använder man ofta det logiska namnet, kallat *programmatic ID* (ProgID), då man skriver kod. När man använder ett ProgID i Visual Basic så översätts det automatiskt till ett GUID. I Visual C++ kan man konvertera från ProgID till GUID med makrot `__uuidof()`.

Innan en applikation (klient till komponenten) kan skapa några COM-objekt måste applikationen initiera COM. Detta gör applikationen genom att anropa API-funktionen `CoInitialize()`. Och innan applikationen avslutar måste den meddela COM att den är färdig och anropa API-funktionen `CoUninitialize()`.

Många av de API-funktioner och gränssnitt som nämns/behandlas i nästa stycke behöver aldrig en programmerare i Visual Basic bry sig om – de anropas och används i bakgrunden.

2.3.1 Hur objekt skapas

För att skapa ett COM-objekt behövs en pekare till COM-klassen för objektet man vill skapa. Ett anrop till COM:s API-funktionen `CoGetClassObject()`, med bl.a. klassens ProgID/ClsID som parameter, returnerar en pekare till COM-klassens gränssnitt `IClassFactory`. Därefter anropas metoden `CoCreateInstance()` (i gränssnittet `IClassFactory`) och tillbaka skickas en gränssnittspekare[8] (*interface pointer*) till ett objekt av klassen. Alla anrop till objektet metoder sker via denna gränssnittspekare.

Ett snabbare sätt att utföra ovanstående är att enbart anropa API-funktionen `CreateInstance()`, d.v.s. metoden utför de två metoderna ovan.

Ett anrop till `CreateInstance()` uppdrar åt COM:s *Service Control Manager* (SCM, uttalas på eng. ”*scum*”) att hitta aktuell klass och skapa objektet (se Figure 2-2, Kirtland). Innan objektet skapas genomför SCM en säkerhetskontroll för att se om den anropande klienten (eller snarare användaren som använder programmet) får lov att skapa objektet över nätverket. Därefter kan SCM, om man önskar, även kontrollera om varje anrop till en metod är tillåten.

2.3.2 Registrering av komponenter

En komponent måste registreras i registret för att SCM ska kunna hitta och instansiera komponenten. Vid registrering skapas, vad som kallas, nycklar i registret som bl.a. innehåller sökvägen till komponenten (t.ex. på lokal disk eller på annan dator). Om komponenten används på samma dator som den utvecklades brukar vissa IDE:er (t.ex. VB) automatiskt registrera komponenten. Om så inte är fallet kan man använda sig av programmet REGSVR32.EXE. För att registrera lokala komponenter i COM-server COMSERVER.DLL skriver man följande i kommandotolken:

```
C:\>REGSVR32 COMSERVER.DLL
```

Om komponenten ska exekvera på en annan dator (server) så måste även datornamn där komponent finns registreras i registret. Eventuellt måste även en mindre programsnutt (*proxy*) vara installerad på klientdator (se avsnitt *Var komponenter exekverar*).

[8] Pekaren är egentligen en pekare till en pekare. Detta för att kunna hantera bl.a. komponenter på andra datorer.

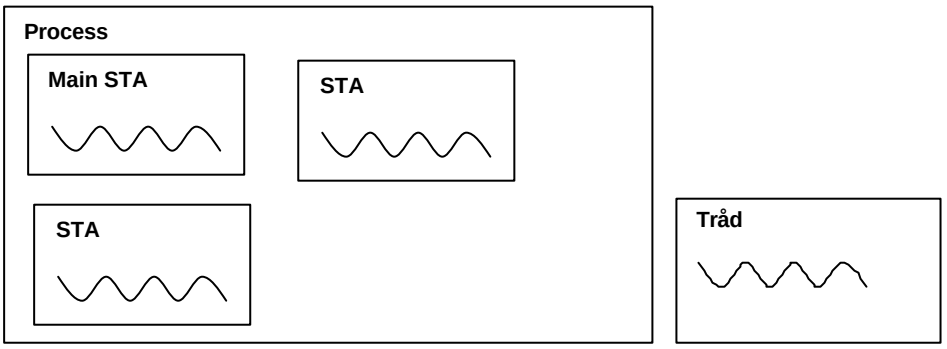
2.4 Hur komponenter exekverar

Då programkod ska exekvera så sker detta i en process och processen tilldelas resurser (minne, CPU-tid, etc.) av operativsystemet. Själva exekveringen inom processen sker i något som kallas trådar (*threads*) och alla trådar inom en process delar på resurser som tilldelats/efterfrågats av processen. En process kan innehålla en eller flera trådar.

I COM associeras varje tråd med en *apartment* som används för att hantera synkronisering av COM-objekts exekvering (i en miljö där flera processer kan exekvera samtidigt – *multi-tasking*). Varje *apartment* kan innehålla en tråd, kallat *single-threaded apartment* (STA), eller flera trådar, kallat *multi-threaded apartment* (MTA). I.o.m. COM+ finns det även en *apartment* som inte har några trådar, kallat *threaded neutral apartment* (TNA), men som lånar trådar från STA och MTA.

2.4.1 STA

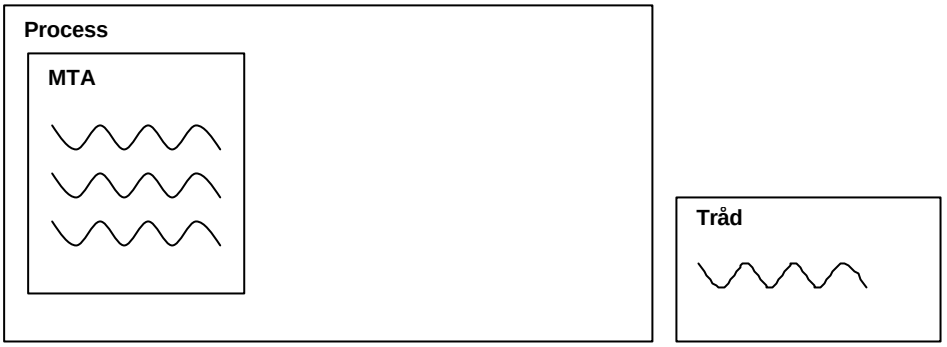
I en STA kan endast en sak ske åt gången, d.v.s. endast en komponents metod kan exekveras åt gången, och alla anrop till alla komponenter i en STA blir därmed synkroniserade. Synkroniseringen löses genom att använda Windows inbyggda meddelandeköer (*message queues*) för fönster[9] som hanterar anrop (meddelande) i den ordning de anländer i kön. Programmering av komponenter som ska exekvera i en STA blir relativt enkel men istället förlorar man i skalbarhet, d.v.s. antalet samtidiga användare av komponenten bör begränsas för att svarstiderna inte ska bli för långa. Varje process kan ha fler STA:er och den första STA:n som skapas brukar kallas Main STA[10].



2.4.2 MTA

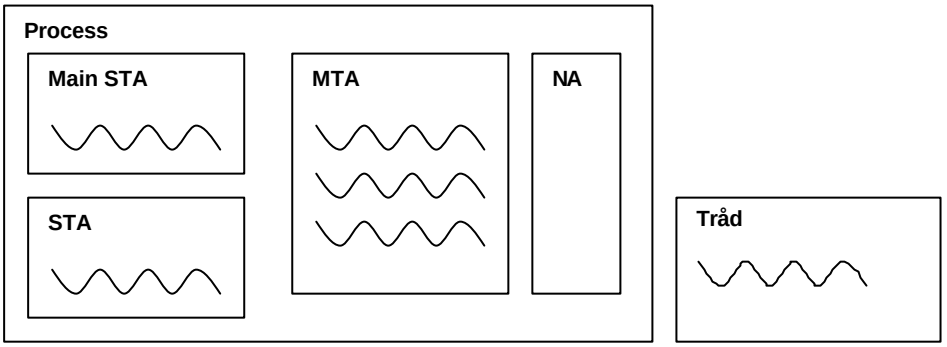
I en MTA sker ingen synkronisering (av anrop av komponenters metoder), d.v.s. det kan förekomma flera anrop av samma eller olika metoder i en eller flera komponenter samtidigt. För komponenter i en MTA måste alltså programmeraren av komponenten själv sköta synkronisering av t.ex. delade COM-objekt och databaser. Programmering av denna typ av komponenter blir mer komplex men i gengäld vinner man i skalbarhet, d.v.s. antalet samtidiga möjliga användare ökar. Varje process kan endast ha en MTA.

[9] Även om komponenten inte använder ett gränssnitt mot användaren så skapas ett dolt fönster.
[10] I Main STA (för varje process) exekverar alla 16-bitars program som är gjorda för Windows 3.x.



2.4.3 TNA

I och med Windows 2000 (och COM+) så finns det även en *threaded neutral apartment* (TNA)[11] och varje process kan bara ha en TNA (se mer under *Modellen Neutral*). I motsatts till STA och MTA så har TNA inga egna trådar – TNA använder sig av trådar i processens STA:er eller MTA.



2.4.4 Applikationer, komponenter och *apartments*

Komponenter i en *apartment* är inte medvetna om komponenter i andra *aparments*, d.v.s. man kan inte[12] dela på t.ex. globala variabler mellan två *apartment*. En komponent i en typ av *apartment* kan inte heller hålla en referens direkt till en komponent i en annan *apartment*. Detta för att synkronisering av komponenter i STA:er ska fungera. För att lösa detta används proxy-/stub-objekt och *marshalling* (se stycke nedan).

Vilken typ av *apartment* som komponenter ska exekveras i anges m.h.a. trådmodeller (*threading models*) när man utvecklar eller registrerar komponenten. Trådmodellen lagras i registret under komponentens nyckel `ThreadingModel` när komponenten registreras (t.ex. `HKEY_CLASSES_ROOT\CLSID\{komponentens GUID}\InprocServer32`).

För applikationer anges typen av *apartment* då COM-miljön initieras m.h.a. anrop av API-funktionerna `CoInitialize()` och `CoInitializeEx()` (se *Mer om Visual C++ och komponenter*).

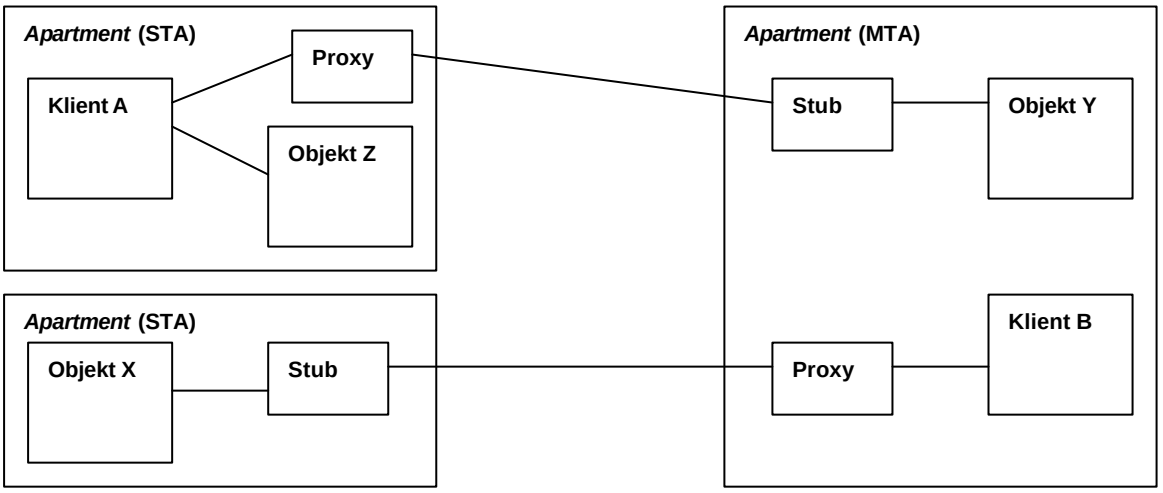
2.4.5 Proxy-/stub-objekt och marshalling

För att COM ska kunna hantera de olika typerna av synkronisering som finns i STA och MTA så tillåter inte COM att en klient i en *apartment* får hålla en direkt referens till en komponent i en annan *apartment*. Det behövs ett sätt att skapa en (indirekt) referens till komponenter i

[11] TNA kallas ibland även för NA eller NTA – *neutral-threaded apartment* – i viss litteratur.
[12] Detta går att komma runt, vilket inte behandlas i dessa sammanfattningar.

andra *apartment*. För detta skapas de två objekten *proxy* och *stub* – ett i respektive apartment – vilka använder *marshalling* för att paketera metodanrop för att skickas mellan objekten.

I klientens apartment skapas ett *proxy*[13]-objekt och som imiterar gränssnittet för den komponenten klienten vill använda. I komponentens *apartment* skapas ett *stub*[14]-objekt, vilken jobbar i par med *proxy*-objektet, och som håller en referens till komponenten som klient vill använda. När sen klienten anropar en metod i komponenten så kommer den göra det mot *proxy*-objektet. *Proxy*-objektet paketerar eventuella parametrar till metoden och skickar anropet till *stub*-objektet som packar upp parametrarna. *Stub*-objektet anropar sen metoden i komponenten och tar hand om resultatet. Resultatet paketeras för att sändas tillbaka till *proxy*-objektet som i sin tur packar upp resultatet och skickar det till klienten.



Marshalling innebär merarbete och leder till att exekveringen tar längre tid. Man bör alltså försöka utveckla komponenter så den exekverar i samma *apartment* som klienten kommer att använda. *Proxy*- och *stub*-objekten genereras av MIDL[15]-kompilatorn.

Marshalling används även då klient och komponent finns i olika processer eller på olika datorer (se *Var komponenter exekverar*).

2.4.6 Trådmodeller

Det finns idag fem trådmodeller för komponenter och de används för att ange vilken typ av *apartment* som komponenten ska exekvera i. Deras namn kan vara något förvirrande och man rekommenderar idag att man inte använder sig av dem. I stället bör man. Detta kan bli ganska svårt då man måste skapa komponenterna och ange någon modell när man utvecklar komponenten ☺.

2.4.6.1 *Single*

Denna modell är ett ”arv” från tiden då endast 16-bitars Windows (version 3.x och tidigare) fanns. I 16-bitars Windows kunde endast en process exekvera åt gången, d.v.s. det fanns bara en ”tråd”. Illusionen att flera program exekverade samtidigt löstes med *tast-switching* – när ett

[13] *Proxy* betyder ombud på svenska. I detta fall agerar *proxy*-objektet ombud för komponent som finns på annan dator och vidarebefordrar trafik mellan anropande applikation och komponent.
[14] *Stub* betyder stump/stubbe på svenska. *Stub*-objektet är den stump med kod som gör kommunikation mellan *proxy*-objektet och komponenten möjlig.
[15] Microsoft IDL (*Interface Definition Language*).

program exekverat färdigt fick nästa program tillgång till processorn. Detta innebär att man inte hade problem som synkronisering av komponenters exekvering och det fanns endast en *apartment*. Komponenter med trådmodellen *single* som exekverar under 32-bitars Windows skapas oftast[16] i den första STA:n som kallas *main STA*.

Denna modell rekommenderas starkt emot och därmed är dess funktion av mindre intresse. Existerande komponenter med denna trådmodell bör skrivas om för minst modellen *apartment*.

2.4.6.2 *Apartment*

Komponenter som är gjorda för trådmodellen *Apartment* kan endast exekvera i en STA. Om anropande applikation exekverar i en STA så kommer även komponenten exekvera i samma STA. Men om applikationen däremot exekverar i en MTA kommer en ny STA att skapas för komponenten att exekvera i. Det senare fallet kommer att innebära att operativsystemet måste utföra *marshalling* för att exekvera komponenten.

Detta är den enda trådmodellen (bortsett från *Single*) som Visual Basic stödjer. Denna trådmodell är dock den som är bäst lämpad att användas för komponenter som ska exekvera i MTS (se senare kapitel) men inte COM+.

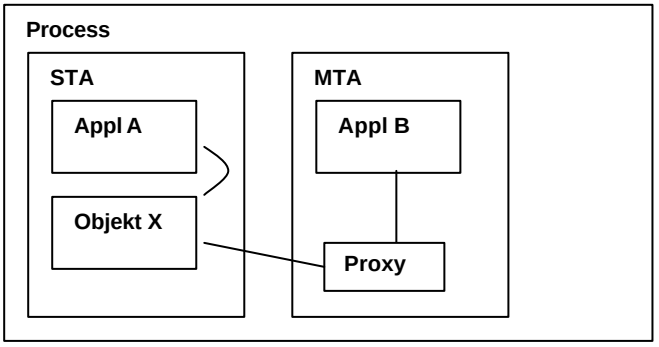
2.4.6.3 *Free*

Trådmodellen *Free* används för komponenter som man vill bara ska exekvera i en MTA. Om anropande applikations exekverar i en MTA så kommer komponenten exekvera i samma MTA och anropen kan ske direkt mot komponenten. Skulle anropande applikation exekverar i en STA så kommer komponenten att skapas i processens MTA och *marshalling* behövs.

2.4.6.4 *Both*

Med hjälp av trådmodellen *Both* så kommer komponenten exekvera i samma (eller samma typ av) *apartment* som anropande applikation. D.v.s. om anropande applikation befinner sig i en STA så kommer även komponenten att exekvera i en STA och motsvarande gäller även för applikationer i en MTA.

Ett undantag från detta är om applikation B vill använda ett existerande (d.v.s. inte skapa ett eget) objekt som applikation A redan skapat. Om en applikation A i en STA har skapat ett objektet X så kommer objekt X finnas i samma STA. När sen applikationen B, i en MTA, vill använda objekt X så kommer *marshalling* att ske för att applikation B ska kunna anropa metoder i objekt X trots att komponenten använder trådmodellen *Both.*



[16] Tyvärr har man som programmerare ingen kontroll över vilken *apartment* som blir *main STA*.

Proxy/stub-paret kan undvikas genom att implementera något som kallas *Free Threaded Marshaler* (FTM) som gör det möjligt att hålla en direkt referens till en komponent i en annan *apartment*. FTM behandlas inte i detta häfte eftersom det bör undvikas (se Eddon/Eddon [99]).

2.4.6.5 *Neutral*

I.o.m. COM+ så introduceras en ny trådmodell – *Neutral* – som exekverar i en TNA. En TNA har som sagt inga egna trådar, utan komponenter som använder trådmodellen *Neutral* använder trådar i klientens *apartment*, oavsett om klienten exekverar i en STA eller MTA.

Komponenter med denna trådmodell använder sig av en annan typ av *proxy*-objekt, *light-weight proxy*, som gör att komponenten aldrig behöver använda sig av ”full” *marshalling*. Dessa lättvikts-proxies fungerar på ett sätt som påminner om FTM.

Ett problem idag är att man inte kan ange trådmodellen *Neutral* i dagens utvecklingsmiljöer – så som Visual Studio 6 – man måste alltså ändra själv i registret.

2.5 Var komponenter exekverar

När komponenter kompileras så kan detta göras till en ”vanlig” exekverbar fil (.EXE) eller till en biblioteksfil (.DLL – *Dynamic Link Library*).[17] En DLL-fil innehåller också exekverbar kod, men DLL:en behöver en process som anropar den exekverbara koden. En sådan process kan t.ex. vara en EXE-fil eller en surrogatprocess (bl.a. MTS under Windows NT eller COM+ i Windows 2000).

En EXE-fil kan (som ni säkert redan känner till) exekvera som ett fristående program men kan också exekvera som en tjänst i Windows NT/2000. En tjänst kan exekvera utan att en (vanlig) användare är inloggad i motsats till ett fristående program som kräver att en inloggad användare startar programmet.[18] DLL-filer måste dock laddas med hjälp av ett anrop från en EXE-fil eller en surrogatprocess. Och eftersom komponenter i distribuerade applikationer tendera till att köras under COM så är de flesta komponenter DLL:er (vilket vi kommer fokusera på i denna sammanfattning).

Om vi utgår ifrån en anropande applikation (eller komponent) så kan en komponent exekvera:

- i samma process som anropande applikation/komponent (*in-process*) – som DLL-fil.
- som en fristående programvara (*out-of-process* eller *local server*) – t.ex. som EXE-fil.
- på en annan dator (*remote server*) – som DLL- eller EXE-fil.

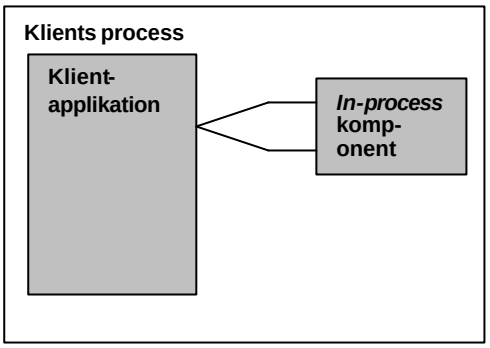
Anropen i kod sker på samma sätt oavsett var komponenten exekverar – COM och DCOM hantera kommunikationen med komponenten – men var komponenten exekverar påverkar bl.a. prestanda.

2.5.1 *In-process* exekvering

När en komponent exekverar *in-process* så exekverar komponenten på samma dator och i samma process som anropande applikation. Komponenten delar på anropande applikations resurser (minne, anslutningar, m.m.). Och eftersom anropen av komponents metoder kan ske direkt mot komponenten så blir svarstiderna relativt snabba.

[17] Valet mellan en EXE- eller DLL-fil brukar ske då projektet för komponenten skapas i programmeringsmiljön (IDE:n), inte vid kompilering. Dock skapas EXE- eller DLL-filen vid kompilering och länkning.
[18] Återigen så är det en sanning med modifikation: man kan starta ett program under tjänsten Task Scheduler med kommandot AT i Windows NT/2000.

En komponent som ska exekvera *in-process* måste skapas som en DLL-fil. Nackdelen med *in-process* exekvering är att om komponenten utför en förbjuden åtgärd så kan den krascha all exekvering i processen.

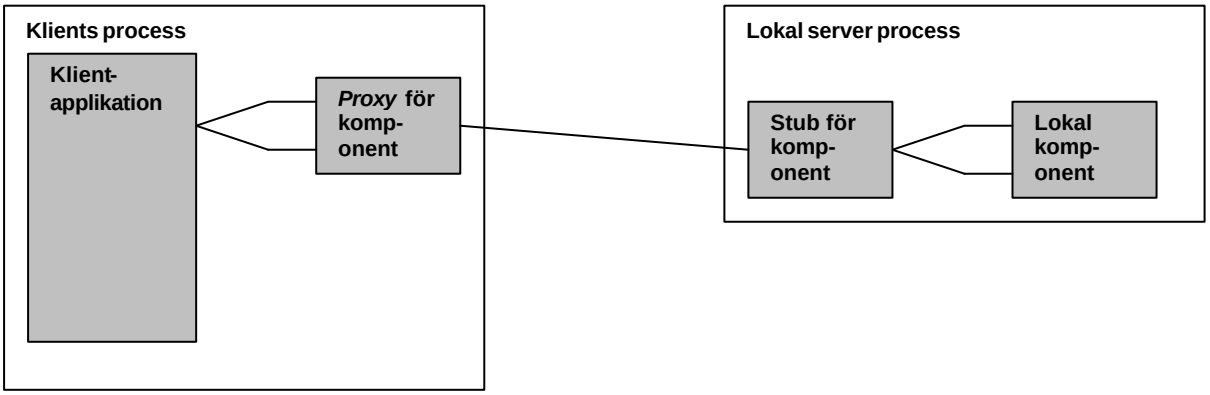


Figur 3 - En in-process *komponent skapad av applikation. Både applikation och komponent exekverar på samma dator och i samma process.*

2.5.2 *Out-of-process* exekvering

När komponenter exekverar på samma dator men utanför anropande applikations process, s.k. *out-of-process* eller *local server*, så kommer komponenterna att ha tillgången till egna resurser. Nackdelen är att anrop av komponentens metoder måste ske via s.k. *marshalling* (se ovan), vilket leder till merarbete och ger därmed längre svarstider. Anropet av metoder är dock lika som vid *in-process* exekvering – COM hanterar denna *marshalling* transparent.

Komponenter som ska exekvera *out-of-process* kan skapas som EXE-filer eller DLL-filer som t.ex. använder MTS som surrogatprocess. Fördelen med *out-of-process* exekvering är att om komponenten utför en förbjuden åtgärd så kan påverkan av applikationens exekvering förhindras genom felhantering i anropande kod.

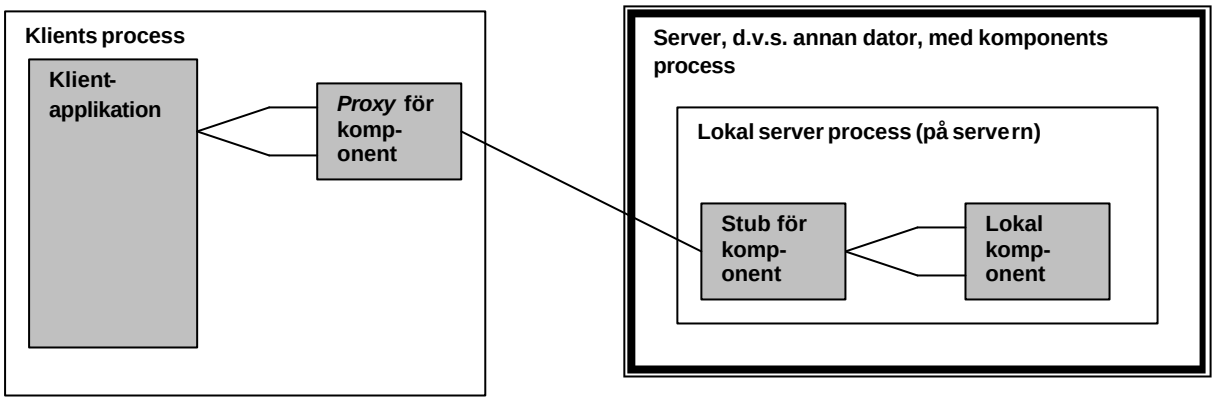


Figur 4 - En out-of-process *komponent skapad av applikation. Både applikation och komponent exekverar på samma dator men i olika processer.*

2.5.3 *Remote server* exekvering

Att komponenten exekverar på en annan dator (*remote server*) är egentligen ett specialfall av *out-of-process* exekvering. *Marshalling* sker på samma sätt som för *out-of-process*, fast kommunikation mellan processer på två datorer blir dock mer omfattande och hanteras av DCOM.

Svarstiden blir (givetvis?) längre då anrop av metoder måste skickas över ett nätverk. Komponenter som ska exekvera på annan dator kan skapas som EXE-filer eller DLL-filer som t.ex. använder MTS som surrogatprocess.



Figur 5 - En remote server *komponent skapad av applikation. Applikationen exekverar på en dator och komponenten på en annan dator. Kommunikationen mellan applikation och komponent hanteras av DCOM.*

2.6 Sammanfattning

Valet av var och hur komponenten exekverar påverkar svarstider genom belastning av operativsystemet och nätverk. Att välja var komponenten exekverar och trådmodell måste avvägas mot vad man vinner/förlorar på att göra på annat sätt. Om t.ex. en komponent exekverar i samma process som *Internet Information Server* (IIS) så kommer anrop av metoder i komponenten ske snabbt. Men om komponenten kraschar så drar komponenten med sig IIS och webbservern kan stanna. Om däremot komponenten exekverar i separat utrymme (*out-of-process*) så kommer anropen av metoder ha längre svarstider och servern belastas mer. Men i fall komponenten kraschar så kommer endast COM-applikationen att krascha och (förhoppningsvis) resten av webbservern fortsätter att fungera.

Komponenter som exekverar i samma process som ett grafiskt gränssnitt kan använda trådmodellen *Apartment* eftersom synkroniseringen blir den samma för bägge. För komponenter som exekverar under MTS så är det trådmodellen *Apartment* som gäller. Det är endast då man har komponenter som inte ska exekverar i ett grafiskt gränssnitt eller under MTS som det är intressant att använda trådmodeller som *Free* och *Both.*

Om vi vill installera komponenterna i COM+ (d.v.s. under Windows 2000/XP) så vill vi helst använda trådmodellen Neutral. Detta innebär att vi måste skriva komponenterna i C++ (eller annat programmeringsspråk) som stödjer trådmodeller för MTA. VB är alltså inte alltid lämpligt under COM+ där vi förväntar oss hundratals eller fler användare (även om det fungerar).

3 Exempel: En första komponent i Visual Basic

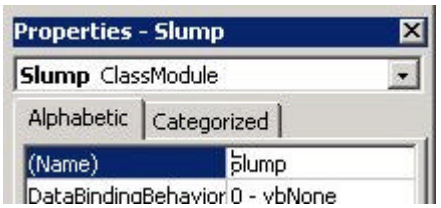
I detta kapitel kommer vi att titta på hur man skapar en enkel komponent. Vi kommer också att titta på ett grafiskt gränssnitt i Visual Basic för att skapa en instans av komponenten och anropa en metod i komponenten. Det ska nämnas att Visual Basic döljer många av detaljerna som har med Microsofts komponentteknologi att göra och att detta exempel endast syftar till att vara ett inledande exempel. Denna komponent använder inte många av COM:s tjänster.

3.1 Komponenten

I detta exempel ska vi göra en slumptalsgenerator som komponent. Komponentens uppgift är att ta fram ett slumptal och sända det tillbaka. Komponenten kallar vi Slump och projektet (COM-servern) KompTest. Du kan senare lägga till klasser i projektet om du vill skapa fler komponenter i denna DLL-fil (COM-server).

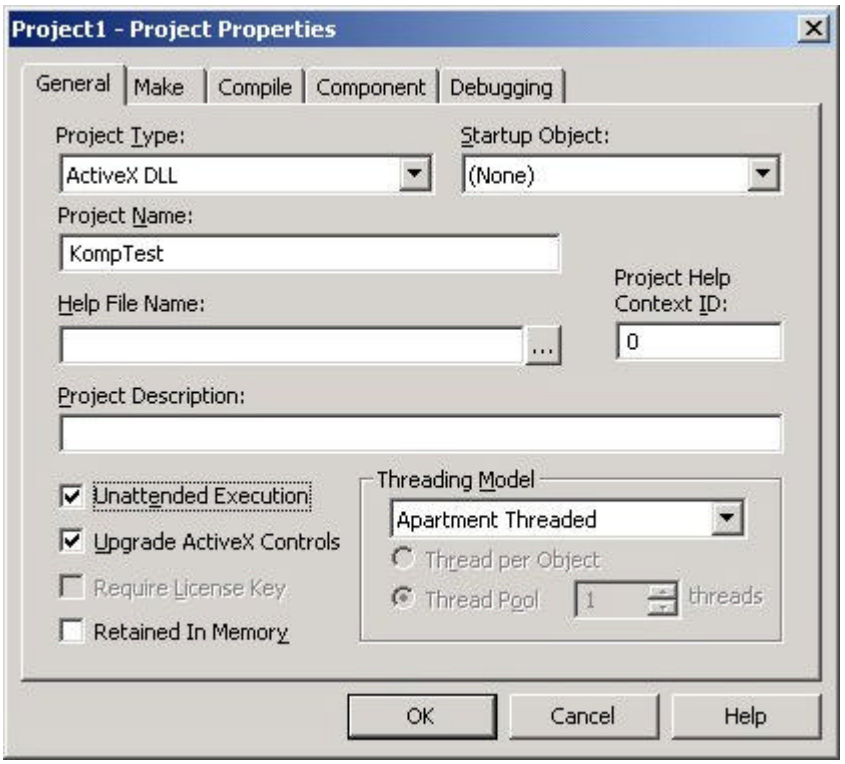
1. Starta Visual Basic.
2. När dialogrutan **New Project** visas – markera **ActiveX DLL** och klicka på **Open** (se bild nedan).

3. Markera klassen `Class1` och ändra dess namn till `Slump` (detta görs i Properties rutan – se till höger). Komponentens namn kommer m.a.o. bli `Slump`.



4. Ändra projektets namn till `KompTest` genom att välja **Project1 Properties…** från Project-menyn och ändra i textrutan **Project Name:**. Därmed kommer komponentens fullständiga namn bli `KompTest.Slump`. Bocka även för **Unattended Execution** (se bild nedan) för att förhindra att man använder sig av metoder som t.ex.

`MsgBox()`. Om vi inte gör det skulle dialogrutor kunna dyka upp på en server där ingen ser dem eller kan stänga dem. I listrutan **Threading Model** visas (eller bör visas) att komponenten kommer exekvera enligt Apartment-modellen. Klicka på OK för att spara och stänga.



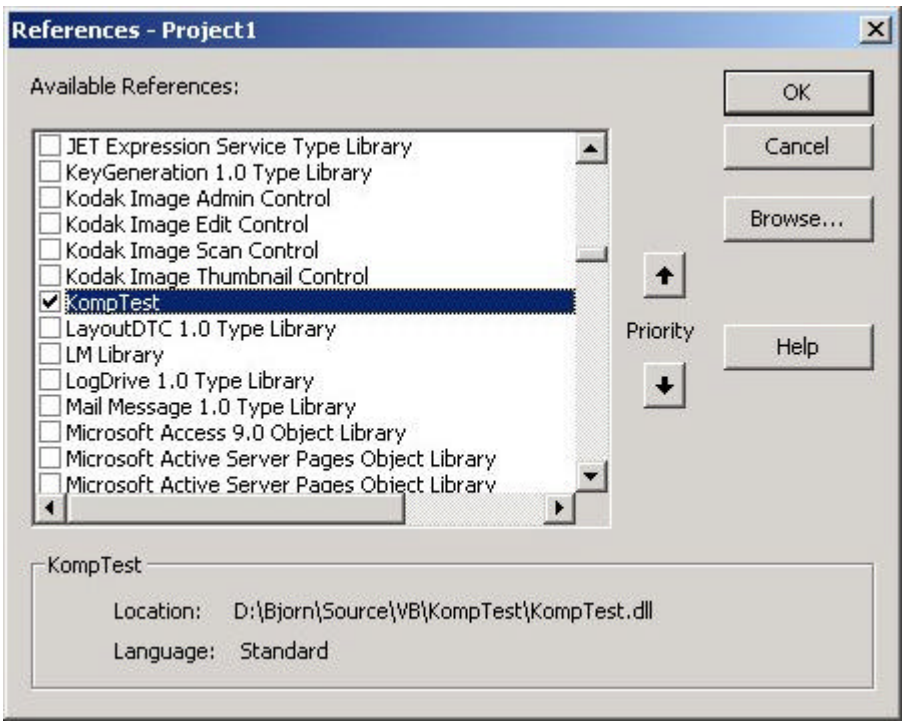
5. Skapa en publik funktion `Slumpa()` genom att skriva följande kod:

```
Public Function Slumpa(max As Integer)
    'Metoden returnerar ett slumptal med maxvärdet max
    Slumpa = Int((max - 0 + 1) * Rnd + 0)
End Function
```

6. Välj **Make KompTest.dll…** från File-menyn. Spara DLL-filen med föreslaget namn (KompTest.Dll).
7. Klart! Du har precis gjort din första komponent. Visual Basics programmeringsmiljö (Integrated Development Enviroment, IDE) registrerar automatiskt komponenten så det är dags att skapa programmet som använder sig av komponenten.

3.2 Testprogrammet

1. Skapa ett nytt projekt genom att välja **New Project** från File-menyn.
2. I dialogrutan **New Project** väljer du **Standard EXE** och klicka på **Open**.
3. Placera en textruta (som ges namnet `Text1`) och en kommandoknapp (som ges namnet `Command1`). Du kanske vill radera textrutans Text-egenskap så att den är tom när programmet körs.
4. Tala om för Visual Basic att du vill använda dig av din nya komponent, d.v.s. ”sätt en referens” till komponentens COM-server. Detta gör du genom att välja **References…** från Project-menyn. När dialogrutan visas letar du upp COM-servern (KompTest) och sätter en bock i rutan framför komponenten (se bild nedan). (Längs ner i dialogrutan ser du sökvägen till komponenten.) Klicka på OK för att stänga dialogrutan.



5. Dubbelklicka på formulärets bakgrund (inte på textrutan eller knappen!) för att öppna kodfönstret och proceduren `Form_Load()`. Fyll i följande kod:

```
Private Sub Form_Load()
    Set objSlump = New KompTest.Slump       'Skapa objekt av komponenten
End Sub
```

6. Lägg till följande kod ovanför ovanstående metod:

```
Option Explicit                  'Be VB att kontrollera att variabler finns
Dim objSlump As KompTest.Slump   'Deklarera variabel av komponent
```

7. Lägg till följande kod nedanför proceduren `Form_Load()` för att ange händelse då knappen klickas på (eller dubbelklicka på kommandoknappen och fyll i koden som saknas nedan):

```
Private Sub Command1_Click()
    Text1 = objSlump.Slumpa(100)
End Sub
```

8. Provkör programmet. När du klickar på knappen kommer ett tal mellan 0 och 100 visas i textrutan.

Observera att VB automatisk visar en lista på möjliga alternativ så fort du skriver ord som `As` eller sätter en punkt efter ett variabelnamn (om variabel är deklarerad och referenser är angivna för projektet). Detta kallar Microsoft för IntelliSense och för att använda sig av markerat alternativ i listan (som dyker upp) trycker man på Tab-tangenten.[19]

Eftersom en COM-server kan ha flera komponenter i sig så kan du lägga till fler klasser i samma Visual Basic-projekt för att lägga till fler komponenter.

[19] En personlig reflektion: Till att börja med är den en väldigt irriterande effekt som stör. Men efterhand som man använder sig av finessen så saknar men den i andra program.

(Denna sida har avsiktligt lämnats blank.)

4 Exempel: En första komponent i C++

I detta kapitel kommer vi att göra om den enkla komponenten i Visual C++ (6.0), d.v.s. den som vi skapade för Visual Basic i förra kapitlet. Vi kommer att återanvända det grafiska gränssnittet i Visual Basic för att skapa en instans av den nya komponenten. I detta kapitel kommer vi att se att C++ inte döljer alla detaljer, d.v.s. koden är mer omfattande. Men det mesta av koden kommer vi använda guider (*wizards*) för att generera.

Observera att detta exempel är gjort för Microsoft Visual C++ 6.0. Se sammanfattningen *Komponenter med COM (och COM+/VC++ 7.0)* för exempel i Microsoft Visual C++ 7.0 (eller C++.NET[20] ☺).

4.1 Komponenten

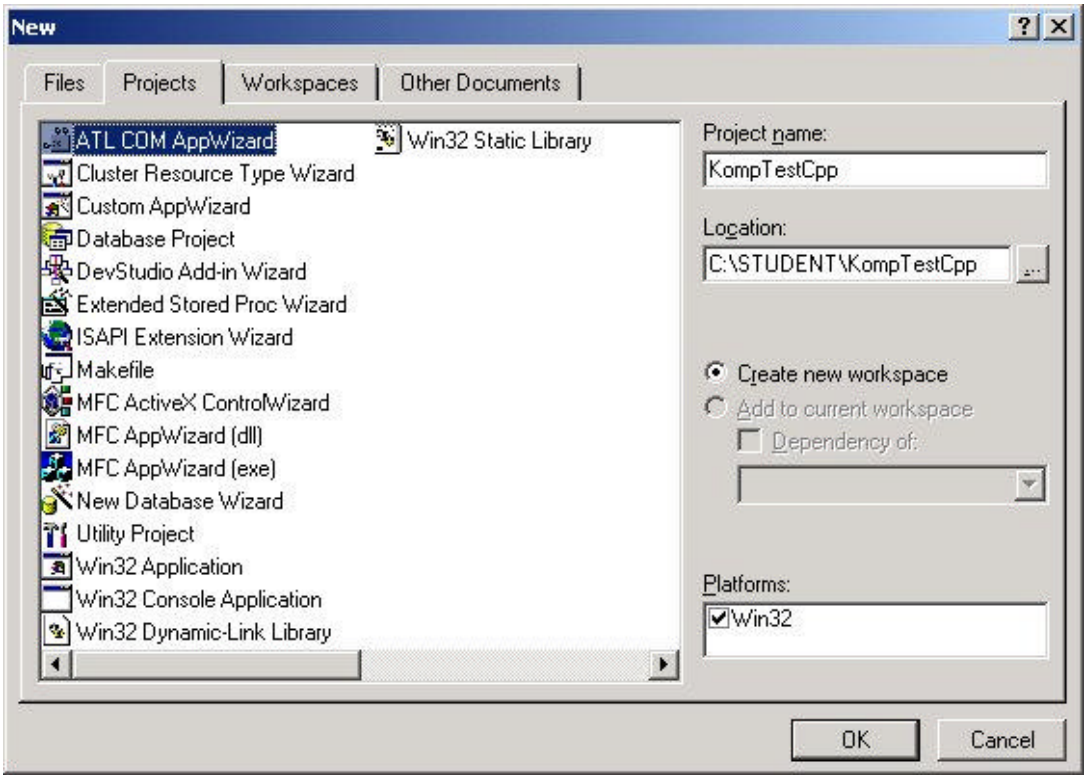
Precis som i förra kapitlet kommer vi göra en slumptalsgenerator som komponent. Komponentens uppgift är att ta fram ett slumptal och sända det tillbaka. Komponenten kallar vi `SlumpCpp` och projektet (COM-servern) `KompTestCpp`.

1. Starta Visual C++.
2. Välj **New…** från File-menyn.
3. Välj fliken **Projects**. Markera ATL COM AppWizard i listrutan och välj mapp (katalog) som du vill att projektets mapp ska skapas i i textrutan **Location:**.[21] [22] Fyll i namnet på projektet (COM-servern) i textrutan **Project name:**, t.ex. `KompTestCpp`. Klicka på OK för att starta ATL-guiden.

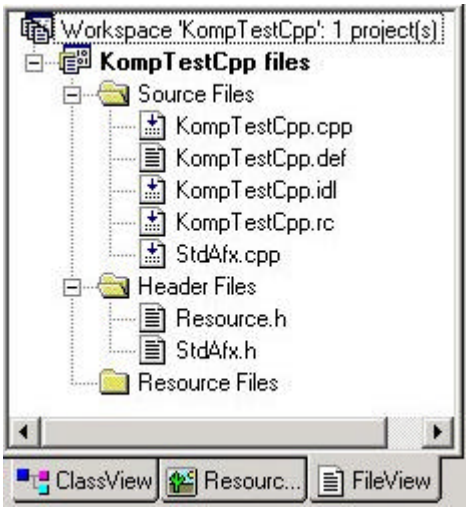
[20] Observera att detta exempel inte använder Managed C++ (d.v.s. exekverar inte under .NET Framework) utan standard C++!

[21] Genom att välja mapp först så kommer VC++ skapa en mapp med samma namn som projektet under den mapp som vi anger i Location:.

[22] När ni jobbar med Visual C++ i nätverk är det lämpligast att jobba mot lokal hårddisk för prestanda – både för er och andra i nätverket. C++ generera en massa små filer när den kompilerar och länkar. Just skapande/öppnande och stängandet av små filer påverkar prestandan på nätverksservrar om många jobbar mot servern samtidigt. Spara därför era projekt i t.ex. C:\Student och när ni är färdiga för dagen kopierar ni alla filer i mappen (**utom** underkatalogerna – dom innehåller ingen källkod) till er hemkatalog.



4. Kontrollera att radioknappen **Dynamic Link Library (DLL)** är markerat under **Server Type**. Lämna kryssrutorna tomma och klicka på **Finish** för att skapa projektet. Klick på OK för att stänga dialogrutan **New Project Information**.



Ni har nu fått ”skalkod” för en COM-server i en DLL-fil. Om ni vill titta på antalet filer och innehållet i dem så klickar ni på fliken **FileView** underst i fönstret *Project Workspace*, kallat projektfönstret i fortsättningen (fönstret till vänster i IDE:n – se bild till höger). I filerna finns främst kod för att ladda och ladda ur DLL-filen samt andra funktioner som en DLL-fil måste stödja. Ta gärna en titt på IDL-filen (för senare jämförelse).

Nästa steg är att skapa klasser som ska bli komponenterna och för detta kommer vi använda fler ATL-guider.

5. Välj fliken **ClassView** i projektfönstret och högerklicka på projektnamnet med fet text längst upp (*KompTestCpp classes* om ni kallade projektet `KompTestCpp`) och välj **New ATL Object…** (eller samma alternativ från Insert-menyn).

6. Markera alternativet **Objects** i listrutan **Category** (till vänster) och sen ikonen **Simple Object** i listrutan **Objects** (till höger). Klicka sen på **Next>** för att starta guiden.

7. Fyll i namnet (`SlumpCpp`) på komponenten i textrutan **Short Name:**. Övriga fält fylls i automatiskt, bl.a. kommer gränssnittet att heta `ISlumpCpp`.

Klicka på fliken **Attributes**. Här kan vi se att trådmodellen är *Apartment* (d.v.s. samma som för Visual Basic-komponenten) och att komponenten kommer stödja *dual interfaces*.

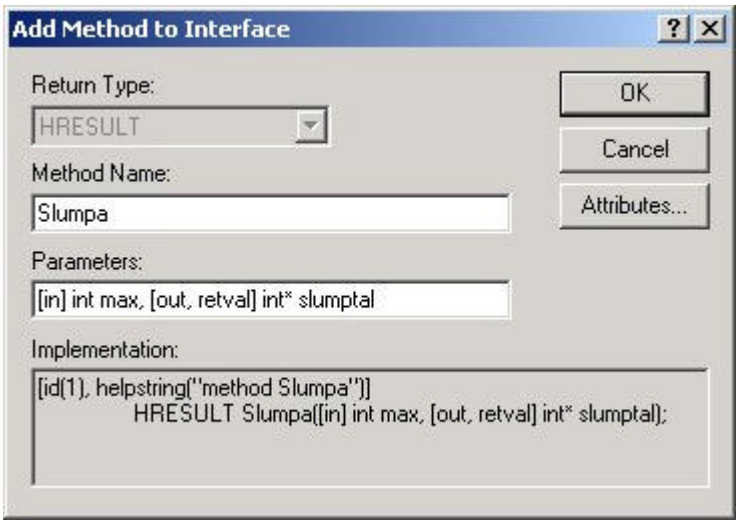
8. Klicka på OK för att stänga dialogrutan **ATL Object Wizard Properties** och för att generera filer samt kod.

Nu har vi alltså skapat komponenten och i detta läge kan vi kompilera komponenten om vi vill. Om vi tittar på fliken **FileView** igen så ser vi att det tillkommit tre filer: *header*-filen och kodfilen (.CPP) för klassen `SlumpCpp` samt en fil (.RGS) för att registrerar komponenten i registret. Tittar vi på IDL-filen igen så ser vi att gränssnittet för komponenten lagts till.

Nästa steg är att lägga till metoden `Slumpa()` och skriva koden för metoden. Först kommer vi använda dialogrutan **Add Method to Interface** för att namnge metoden och ange parametrar.

9. Högerklicka på gränssnittet, *ISlumpCpp* om du namngav komponenten SlumpCpp, och välj **Add Method…**.

10. Fyll i namnet `Slumpa` i textrutan **Method Name:** och `[in] int max, [out, retval] int* slumptal` i textrutan **Parameters:** för att ange att parametern `max` endast behöver skickas in samt att slumptalet kommer att returneras i parametern `slumptal`. Komponenters metoder i C++ returnerar en variabel av typen HRESULT vilket gör att vi måste skicka tillbaka slumptalet via en parameter (`slumptal`) till funktionen som är deklarerar som en pekare (se mer under *Mer om C++ och komponenter*).



11. Expandera först klassen *CSlumpCpp* under fliken **ClassView** och sen gränssnittet *ISlumpCpp* under klassen (se bild till höger). Dubbelklicka på metoden `Slumpa()` (med den rosa ikonen framför) för att visa koden för metoden. (Om vi går direkt på gränssnittet och dubbelklickar på metoden med den gröna ikonen framför så visa Visual C++ metodens definition i IDL-filen.)

12. Skriv av koden nedan så metoden ser ut enligt följande:

```
STDMETHODIMP CSlumpCpp::Slumpa(int max, int *slumptal)
{
   // TODO: Add your implementation code here
   srand( (unsigned)time( NULL ) );   //skapa ett frö

   *slumptal = rand() % (max + 1);   //returnera slumptalet

   return S_OK;
}
```

13. Lägg till `#include <time.h>` högst upp i filen `SlumpCpp.cpp` under raden `#include "SlumpCpp.h"`.

14. Kompilera DLL-filen genom att välja **Build CompTestCpp.dll** från Build-menyn (eller tryck F7 på tangentbordet ☺). Kontrollera att följande visas i resultatfönstret längst ner i IDE:n (*Performing registration* är ”nyckelorden” då komponenten nu blivit registrerad i registret och kan användas direkt) :

```
--------------------Configuration: KompTestCpp - Win32 Debug--------------------
Compiling...
```

```
SlumpCpp.cpp
Linking...
Performing registration

KompTestCpp.dll - 0 error(s), 0 warning(s)
```

15. Komponenten är klar!

4.2 Testprogram i Visual Basic

För testprogrammet kommer vi använda Visual Basic igen och det klientprojekt vi skapade i förra kapitlet. Det ända vi behöver gör är att ändra referensen till den nya DLL-filen med komponenten i och göra smärre förändringar i formulärets kod.

1. Ändra referensen till den nya DLL-filen (COM-servern) genom att välja **References…**från Project-menyn. Observera att den nya COM-servern heter **KompTestCpp 1.0 Type Library** (se bild nedan).

2. Ändra koden för formuläret så att den blir enligt följande (d.v.s. ändra 2 `KompTest.Slump` till `KOMPTESTCPPLib.SlumpCpp`):

```
Option Explicit
Dim objSlump As KOMPTESTCPPLib.SlumpCpp

Private Sub Command1_Click()
    Text1 = objSlump.Slumpa(100)
End Sub

Private Sub Form_Load()
    Set objSlump = New KOMPTESTCPPLib.SlumpCpp
End Sub
```

3. Kör programmet och klicka på kommandoknappen.

Som vi ser av koden så ger Visual C++ (som standard) COM-servern ett ProgID som motsvaras av namnet på projektet och ordet ”Lib” (i vårt exempel `KOMPTESTCPPLib`) men låter komponenten behålla sitt namn (`SlumpCpp`). Vi märker också att man bara skickar ett argument till metoden `Slumpa()`, inte två som implementationen av metoden i C++ har. Genom att ange parametern `slumptal` som `[out, retval]` så kommer bl.a. Visual Basic och skriptspråk betrakta detta som resultatet från metoden. Det egentliga resultatet från C++-funktionen (`HRESULT`) kopieras till objektet `Err` i Visual Basic (se mer under *Mer om C++ och komponenter*).

```
Private Sub Command1_Click()
    Text1 = objSlump.Slumpa(100)
    MsgBox "Felnummer: " + Str(Err.Number)
End Sub
```

(Varför händer ibland inget när man trycker på knappen? Svaret finns i senare kapitel. ☺)

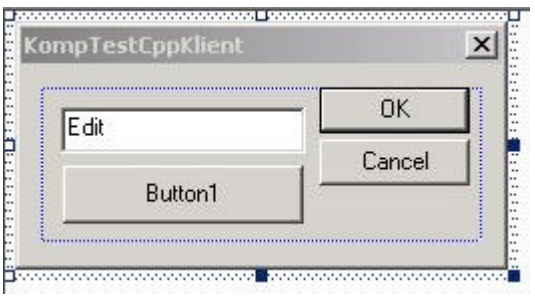
4.3 Testprogram i Visual C++ (”överkurs”)

Detta andra testprogram kommer vi göra i Visual C++. Detta är ”överkurs” men har tagits med för dem som vill ha en introduktion till grafiska gränssnitt i Visual C++. Exemplet kommer dock visa hur man sätter en referens till en komponent i Visual C++ samt hur man skapar en instans av komponenten i Visual C++.

1. Starta ett nytt projekt men välj ett *MFC AppWizard (exe)*-projekt. Kallar programmet för t.ex. KompTestCppKlient (i textrutan **Project name:**). Klicka på OK.
2. Klicka på radioknappen **Dialog based** och sen **Next>**.
3. Avmarkera kryssrutan **About box** och klicka på **Next>**.
4. Klicka på **Next>** i nästa dialogruta (**Step 3 of 4**).
5. Klicka på **Finish** och sen OK.

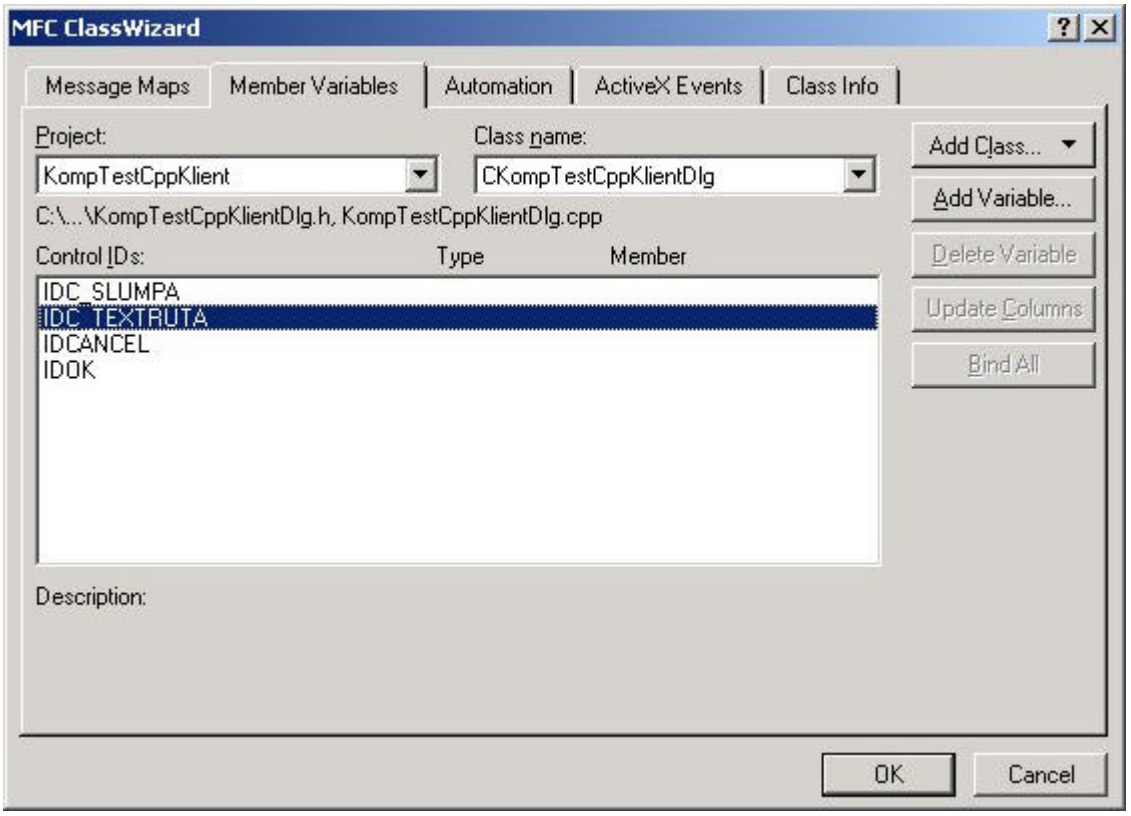
Ni har nu ett fullt fungerande program som visar en dialogruta med OK- och Cancel-knapp. Nästa steg är att lägga till en textruta och en kommandoknapp. Till textrutan kommer vi koppla en medlemsvariabel (m_Slumptal) som kommer uppdatera textrutan automatiskt när vi klickar på kommandoknappen. Vi kommer se att det är lite mer jobb i Visual C++ att skapa ett grafiskt gränssnitt än det är i Visual Basic.

6. Om formuläret för dialogrutan inte visas i IDE:n: välj fliken **ResourceView** i projektfönstret (till vänster) och expandera Dialog-grenen för att sen dubbelklicka på formulärobjektet (med namnet `IDD_KOMPTESTCPPKLIENT_DIALOG` om ni valde namnet `KompTestCppKlient` som namn på projektet). Formuläret för dialogrutan kommer att visas i resurseditorn (*resource editor*) som fungerar på ett liknande sätt som formulärfönstret i Visual Basic.
7. Klicka på etiketten med texten ”TODO: Place…” i och ta bort den genom att trycka på Delete-knappen (på tangentbordet ☺). Det går bra att ta bort även knapparna eftersom dom inte kommer fylla någon funktion (annat än att dialogrutan kommer stängas).
8. Klicka på ikonen för textruta (*edit box*) i verktygsfältet (ikonen med ”**ab**|” på) och rita en textruta på formuläret (se bild till höger).
9. Klicka på ikonen för kommandoknappen (*button*) och rita kommandoknappen på formuläret. Det går bra att göra fönstret lite minde (se bild till höger).

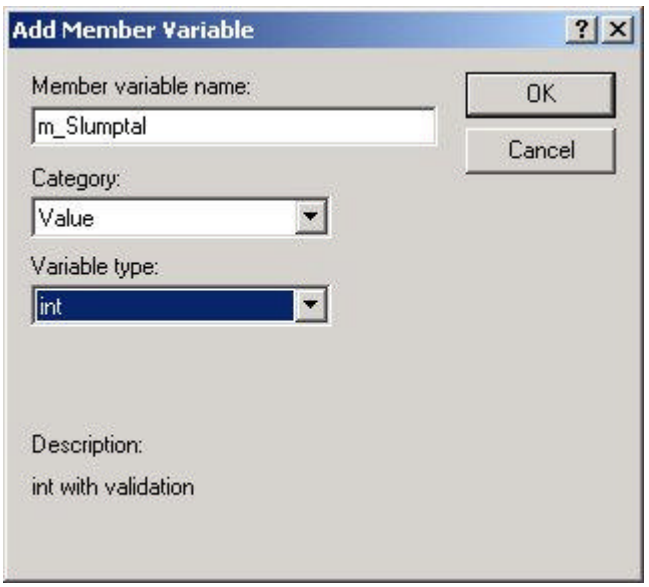


10. Högerklicka på knappen och välj **Properties…** från menyn för att visa egenskapsdialogen. (Klicka gärna på ”häftstiftet” längst upp till vänster i dialogrutan som visas så att den stannar kvar – se bild nedan.) Ändra till `IDC_SLUMPA` i komborutan **ID:** och till `Nytt slumptal` i textrutan **Caption:** (se bild nedan). (Prefixet `IDC_` används för att identifiera kontrollen och skilja den från andra variabler i projektet.)

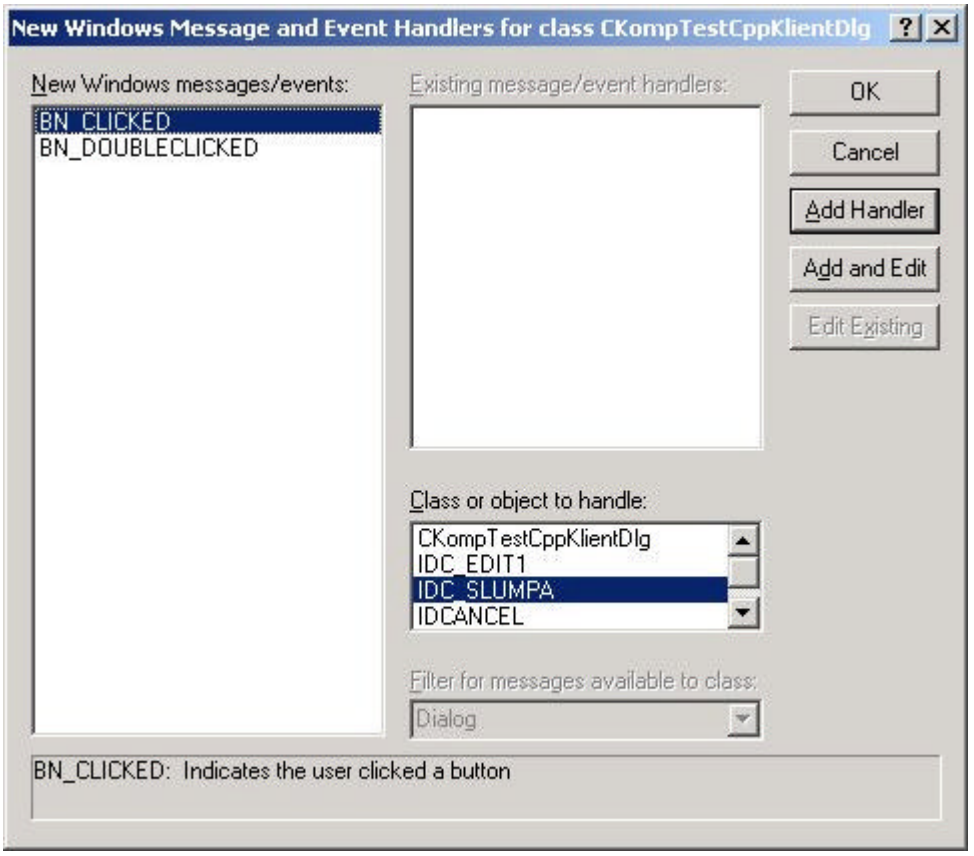
11. Visa egenskapsdialogen för textrutan (t.ex. genom att bara klicka på textrutan om ”häftstiftet” är ”fastsatt” i förra punkten). Ändra innehållet i komborutan **ID:** till `IDC_TEXTRUTA`. (Om ni klickade på ”häftstiftet” så kan ni stänga egenskapsdialogen nu.)
12. Högerklicka på textrutan och välj **ClassWizard…** från menyn. Välj fliken **Member Variables**.
13. Markera IDC_TEXTRUTA och klicka på knappen **Add Variable…**.



14. Fyll i namnet `m_Slumptal` för medlemsvariabeln (**Member variable name:**) och typen till `int` (**Variable type:**). Kontrollera även att komborutan **Category:** visar `Value`. Klicka OK för att stänga dialogrutan **Add Member Variable** och sen OK för att stänga dialogrutan **MFC ClassWizard.**.

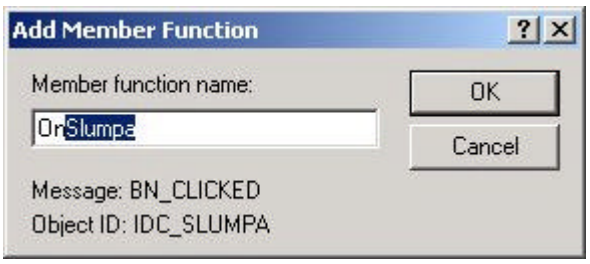


15. Högerklicka på den nya kommandoknappen (`IDC_SLUMPA`) och välj **Events…** från menyn som visas. Kontrollera att dialogrutan ser ut enligt följande bild:



Klicka på knappen **Add and Edit**.

16. Godkänn namnet på händelsefunktionen (`OnSlumpa`) genom att klicka på OK.



17. Ersätt kommentaren (`//TODO:…`) med följande kod:

```
int max = 10;         //Maxvärde på slumptal
int slumptal = 0;     //Variabel att returnera slumptalet från komponent

CoInitialize(NULL);  //Initiera COM-miljön

KOMPTESTCPPLib::ISlumpCpp* pSlump = NULL;    //Pekare till komponents gränssnitt

   //Skapa objekt
CoCreateInstance(__uuidof(KOMPTESTCPPLib::SlumpCpp),
        NULL, CLSCTX_INPROC_SERVER,
        __uuidof(KOMPTESTCPPLib::ISlumpCpp),
        reinterpret_cast<void**>(&pSlump));

   //Ta fram ett nytt slumptal och returnera genom m_Slumptal
pSlump->raw_Slumpa(max, &m_Slumptal);
pSlump->Release();    //Minska referensräknare så att COM kan förstöra instansen

CoUninitialize();     //Meddela COM att vi är färdiga med COM-miljön
UpdateData(FALSE);    //Uppdatera kontroller på formuläret
```

18. För att koden ska fungera måste vi importera komponentens *type library*, vilket vi gör med följande kod som placeras längst upp (i samma fil som funktionen ovan) under den sista `#include`-satsen (ändra sökvägen om ni placerat komponenten annanstans):

```
#import "C:\Student\KompTestCpp\Debug\KompTestCpp.dll" no_namespace
```

19. Kompilera projektet genom att välja **KompTestCppKlient.exe** från Build-menyn för att kontrollera att allt är rätt avskrivet. Klart – kör programmet!

4.3.1 Förklaring till koden i metoden `OnSlumpa()`

Här förklaras det som inte har med COM att göra – det COM-relaterade förklaras i nästa exempel. De två variablerna som deklareras används i anropet till komponentens metod och är vanlig C++-kod. Det är god sed att initiera variablerna för att undvika fel vid exekvering, d.v.s. att de innehåller ett värde så att vi kan använda variablerna direkt. I denna metod får inte variabeln `max` innehålla 0 för då kommer division med noll ske i (den enkla!) komponenten (vilket vi alla vet inte går ☺).

```
int max = 10;         //Maxvärde på slumptal
int slumptal = 0;     //Variabel att returnera slumptalet från komponent
```

Nästa rad fortsätter med att initiera COM-miljön (se nästa kapitel).

```
CoInitialize(NULL);  //Initiera COM-miljön
```

Här näst deklareras en variabel som pekare till gränssnittet för komponenten som ska skapas. Variabeln sätts till `NULL` för att vara på säkra sidan (se mer nästa kapitel).

```
KOMPTESTCPPLib::ISlumpCpp* pSlump = NULL;    //Pekare till komponents gränssnitt
```

För att skapa objektet används funktionen `CoCreateInstance()` (se nästa kapitel).

```
CoCreateInstance(__uuidof(KOMPTESTCPPLib::SlumpCpp),
        NULL, CLSCTX_INPROC_SERVER,
        __uuidof(KOMPTESTCPPLib::ISlumpCpp),
        reinterpret_cast<void**>(&pSlump));
```

Nästa steg är att anropa metoden `Slumpa()` i komponenten. Här använder vi variabeln `max` för att ange maxvärdet på slumptalet och medlemsvariabeln `m_Slumptal` för att returnera slumptalet från metoden (se mer nästa kapitel).

```
    //Ta fram ett nytt slumptal och returnera genom m_Slumptal
pSlump->Slumpa(max, &m_Slumptal);
```

När vi är klara med COM-objektet ska vi meddela den genom anropa metoden `Release()`(se nästa kapitel).

```
pSlump->Release();    //Minska referensräknare så att COM kan förstöra instansen
```

Sist av allt (vad gäller COM-miljön) talar vi om för COM att vi är klara genom att anropa `CoUninitialize()` (se nästa kapitel).

```
CoUninitialize();    //Meddela COM att vi är färdiga med COM-miljön
```

Sista raden används för att uppdatera kontrollerna på formuläret och har med användargränssnittet att göra, d.v.s. inget med COM att göra.

```
UpdateData(FALSE);    //Uppdatera kontroller på formuläret
```

Detta sista exempel av användargränssnitt i C++ bör kunna användas för att visa på att det är mindre jobb att använda Visual Basic för användargränssnitt ☺. Något som också saknas helt är felhantering – mer om detta i nästa kapitel.

5 Mer om Visual C++ och komponenter

Detta kapitel fortsätter från förra kapitlet med att förklara hur man skapar komponenter i Visual C++. Kapitlet kommer också behandla detaljer som skiljer utveckling av komponenter i Visual C++ från de i Visual Basic. Bl.a. stöder Visual C++ fler trådmodeller och kan i vissa fall vara lämpligare att använda för en del komponenter. Komponenter skrivna i C++ tenderar också till att exekvera snabbare än program skrivna i Visual Basic (oavsett trådmodell). Nackdelen med Visual C++, jämfört med Visual Basic, är att det ofta behövs mer kod och en bättre förståelse av COM för att skapa komponenter. Detta underlättas dock av ATL-guiderna i Visual C++.

Att använda komponenter i Visual C++ kräver också en bättre förståelse av COM men i vissa avseende kan man få kodens syntax i C++ att påminna den i Visual Basic.

5.1 Visual C++ som klient mot komponenter

I detta exempel kommer vi att skapa ett program för kommandotolken för att visa och förklara hur vi använder Visual C++ som klient mot komponenten vi skapade i förra kapitlet. Därmed kan vi fokusera på koden för att skapa komponenten utan att störas av kod för gränssnittet. Komponenten kommer att skapas i programmet funktion main(), men den skulle lika väl kunna skapas i en annan funktion. D.v.s. koden som kommer skrivas i main() skulle likaväl kunna skrivas i vilken funktion som helst i C++.

När vi skapar projektet så anger vi **Win32 Console Application** som projekt och sen markerar vi alternativet **A "Hello_World!" application**. Därmed får den kod och de filer vi behöver för att kunna köra programmet.

5.1.1 Inkludera komponenters definition

Det första som behövs för att använda en komponent i Visual C++ är att inkludera komponentens definition av gränssnittet. Definitionen finns i komponentens *header*-fil (`projektnamn.h`) och genereras av MIDL utifrån komponentens IDL-fil. För att inkludera komponenten Slumpa så anger vi namnet på *header*-filen för projektet (COM-servern) vi skapade i förra kapitlet. I detta exempel skapas detta program i en katalog på samma nivå i katalogstrukturen som den katalog där komponenten skapades, d.v.s. sökvägen till komponentens *header*-fil blir `..\Projektnamn\Projektnamn.h`. Nedan visas koden som guiden skapat åt oss med och där komponentens *header*-fil har inluderats med direktivet `include`. Vi har även inkluderat *header*-filen `iostream.h` för att kunna använda `cout` för utskrifter.

```
#include "stdafx.h"                          //Lades till av guiden
#include <iostream>                         //För att använda cout
#include "..\KompTestCpp\KompTestCpp.h"     //Länka in header-filen för komponenten

using namespace std;

int main(int argc, char* argv[])            //Resterande kod lades till av guiden
{
   printf("Hello World!\n");                //Kommer att raderas...
   return 0;
}
```

I Visual Basic motsvaras inkluderingen av *header*-filen med att man anger referenser för projektet (till DLL-filen), d.v.s. i dialogrutan **Project References** som nås med menyalternativet **References…** från Project-menyn (i Visual Basic).

5.1.2 Initiera COM

Som nämnts tidigare så måste man initiera COM-miljön (i klienter) innan man kan börja använda komponenter. Det gör man som sagt med ett anrop till API-funktionen `CoInitialize()`, som ska balanseras med ett motsvarande anrop till `CoUninitialize()` när man är färdig med komponenterna. Som enda parameter till `CoInitialize()` skickar man `NULL` då parametern idag inte används (den har alltså använts tidigare).

```
void main(int argc, char* argv[]) {
   CoInitialize(NULL);   //Initialisera COM för att exekverar i en STA

      //Programmets kod

   CoUninitialize();     //Meddela COM att vi är färdig med komponenterna
   return 0;
}
```

I applikationer för COM+ bör man inte använda `CoInitialize()` utan `CoInitializeEx()`.[23] Precis som för `CoInitialize()` ska första parametern till `CoInitializeEx()` vara `NULL` och som andra parameter anger man vilken typ av trådmodell man vill använda – STA (`COINIT_APARTMENTTHREADED`) eller MTA (`COINIT_MULTITHREADED`). Anropet av `CoUninitialize()` fungerar annars på samma sätt.

```
void main() {
  CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);  //Exekverar program i en STA
  //CoInitializeEx(NULL, COINIT_MULTITHREADED);    //Exekverar program i MTA

    //Programmets kod

  CoUninitialize();
  return 0;
}
```

Eftersom vår komponent har trådmodellen *Apartment* så bör vi välj det första alternativet (det som inte har kommenterats bort ovan). Får vi felmeddelandet `'CoInitializeEx' : undeclared identifier` från kompilatorn så måste vi ändra en rad i filen `StdAfx.h`. Leta upp en rad som påminner om första raden i exempel nedan och ersätt med raden nedanför den.[24]

```
//#define WIN32_LEAN_AND_MEAN      //Rad att ersätta
#define _WIN32_DCOM                //Rad att ersätta med
```

Som Visual Basic-programmerare behöver man inte bry sig om att anropa `CoInitialize()` och `CoUninitialize()` – detta sker helt i bakgrunden.

[23] Skulle man ändå använda `CoInitialize()` så kommer COM+ att anropa `CoInitializeEx()` med andra parametern satt till `COINIT_APARTMENTTHREADED`.
[24] Vad som står efter `#define` kan variera beroende på vilken typ av klient (till komponent) vi skapar. Men oftast börjar raden med `#define WIN32_xxx`.

5.1.3 Skapa objekt

Innan vi kan skapa COM-objekt behöver vi en variabel att lagra gränssnittspekaren i. I koden nedan deklarerar vi några variabler till som vi bl.a. behöver för att hantera resultatet från funktionen i komponenten.

```
int main(int argc, char* argv[])
{
   int i = 0;        //Räknare för loop
   int max = 10;     //Maxvärde för slumptal (=max+1)
   int slumptal = 0; //Variabel för att hantera returvärdet.
   ISlumpCpp* pSlump = NULL;   //Pekare till gränssnitt

   CoInitialize(NULL);   //Initialisera COM-miljön

    //Programmets kod

  CoUninitialize();
  return 0;
}
```

För att skapa ett COM-objekt används API-funktionen `CoCreateInstance()`. Funktionen tar fem parametrar:

Parameter	**Förklaring**
ClsID för COM-klassen	Detta GUID är ClsID för klassen som implementerar gränssnittet (i parameter fyra). Makrot `__uuidof()` kan användas för att konvertera ett ProgID för klassen till ett GUID.
En pekare till ett objekt som detta objekt är en aggregerad del av	Om objektet inte är aggregerat så skickas `NULL`. (Aggregerade objekt behandlas inte i denna sammanfattning då det inte är så vanligt förekommande.)
Var komponent ska exekvera	Om komponenten ska exekvera *in-process*, *out-of-process* eller på en annan dator (se beskrivning av konstanter nedan).
IID för komponentens gränssnitt	Makrot `__uuidof()` kan användas för att konvertera ett ProgID för gränssnittet till ett GUID som vi vill ha returnerad i parameter nedan.
Gränssnittspekare	En pekarvariabel för att returnera gränssnittspekaren för objektet.

Det finns konstanter deklarerade för tredje parametern i funktionen `CoCreateInstance()` som kan användas för att tala om för COM var man vill skapa instansen:

- `CLSCTX_INPROC_SERVER` – komponenten är en DLL-fil och ska exekverar i klientens process.
- `CLSCTX_LOCAL_SERVER` – komponenten är t.ex. en EXE-fil som ska exekverar i en separat process på samma dator.
- `CLSCTX_REMOTE_SERVER` – komponenten exekverar på en annan dator.
- `CLSCTX_SERVER` – komponenten kan vara en av de tre ovan. Gränssnittspekare till den först tillgängliga komponenten kommer returneras.
- `CLSCTX_ALL` – komponenten kan vara alla av ovanstående.

Observera att komponenten måste vara registrerad att exekvera på det ställe vi anger med någon av konstanterna ovan. D.v.s. har vi registrerat en komponent i en DLL-fil lokalt på datorn så kan vi använda `CLSCTX_INPROC_SERVER` men inte `CLSCTX_REMOTE_SERVER`.

Eftersom vår komponenten är en DLL så kan vi exekvera den *in-process* och använder därför konstanten `CLSCTX_INPROC_SERVER`.

```
CoInitialize(NULL);   //Initialisera COM-miljön

hr = CoCreateInstance(__uuidof(SlumpCpp),
                      NULL,
                      CLSCTX_INPROC_SERVER,
                      __uuidof(ISlumpCpp),
                      reinterpret_cast<void**>(&pSlump));

   //Programmets kod

CoUninitialize();
```

Metoden `CoCreateInstance()` kan endast skapa objekt på den lokala datorn. Om man istället vill skapa ett objekt på en annan dator måste man använda metoden `CoCreateInstanceEx()` (se exempel i sammanfattningen *Komponenter med DCOM/MTS*). Denna metod kan även returnera gränssnittspekare till flera olika gränssnitt samtidigt. Detta kan innebära färre tidskrävande anrop över t.ex. nätverket om man vill använda sig flera gränssnitt i objektet. (Se hjälpen för Visual Studio för mer information.)

Dessa två metoder för att skapa objekt anropar själva en hel del metoder, bl.a. `AddRef()` i objektets gränssnitt `IUnknown`. Därför behöver vi som programmerare i Visual C++ aldrig anropa denna viktiga metod. Däremot får man aldrig glömma att anropa `Release()` när man är klar med objektet – en gång mot alla gränssnitt man har pekare till! Annars kommer minnet att ”läcka” då objekten inte kommer att förstöras och därmed tas bort från minnet.

I Visual Basic anropas funktionerna `CoCreateInstance()` och `AddRef()` när man använder New-operatorn och `Release()` när vi sätter variabeln till `Nothing` (eller när variabelns räckvidd tar slut, t.ex. när exekvering av metod slutar).

```
Set objSlump = new KompTest.Slump   'Visual Basic anropar i bakgrunden
                                    '  CoCreateInstance() och AddRef().
 '...
Set objSlump = Nothing              'VB anropar Release() i bakgrunden
```

5.1.4 Anrop av objekts metoder

Eftersom variabeln som ”innehåller” komponenten är en pekare till ett gränssnitt så måste vi använda piloperatorn (”->”) för att anropa komponentens metoder. I exemplet nedan anropar vi metoden `Slumpa()` i variabeln `pSlump`. Slumptalet returneras genom variabeln `slumptal` (som vi måste skicka referensen till med &-operatorn). Vi loopar tio gånger och i varje loop skriver vi ut slumptalet samt pausar i 1000 millisekunder[25].

```
for(i = 0; i < 10; i++)
{
```

[25] Som slumptalsgeneratorn fungerar i förra kapitlet så genereras fröet genom att läsa av datorns tid (sekunden) vilket gör att om vi inte pausar i minst en sekund så kommer de flesta slumptalen vara de samma. Prova gärna själv genom att kommentera bort raden `Sleep(1000);`.

```
        pSlump->Slumpa(max, &slumptal);           //Anropa metod i komponent
        cout << "Slumptal: " << slumptal << endl; //Skriv ut slumptal
        Sleep(1000);                              //Pausa i 1000 millisekunder
    }
```

5.1.4.1 Använda accessmetoder

I Visual Basic implementeras (lämpligen) egenskaper med privata variabler och med accessmetoder i klasser. Metoderna får samma namn som egenskapen och anropas i Visual Basic med dessa namn.

I Visual C++ får dessa egenskaper ett lite annorlunda namn – de ges prefixen ”`put_`” och ”`get_`”. Om vi skriver om komponenten från förra kapitlet och ändrar till egenskaper så skulle vi kunna få följande kod:

```
    pSlump->put_MaxVarde(max)                  //Sätt värde på egenskapen MaxVarde
    pSlump->InitieraSlumptal()                 //Initiera slumptalsgenerator
    for(i = 0; i < 10; i++)
    {
        pSlump->get_Varde(&slumptal);           //Hämta värde på egenskapen Varde
        cout << "Slumptal: " << slumptal << endl; //Skriv ut slumptal
    }
```

I exempel ovan måste vi först sätta egenskapen för maxvärdet (`MaxVarde`)och initierade komponentens frö (`InitieraSlumptal()`) innan vi kan hämta värdet på egenskapen `Varde` i komponenten.

5.1.5 Fel vid exekvering av metoder

Alla metoder i COM returnerar ett 32-bitars värde kallat `HRESULT` och som visar på varför en metod misslyckades eller hur den lyckades. D.v.s. alla metoder i komponenter som utvecklas i C++ bör returnera ett HRESULT-värde. Om metoden lyckades brukar man returnera ett lyckat resultat med konstanten S_OK.

Som påpekats ett antal gånger tidigare så döljer Visual Basic många detaljer för programmeraren. Och för att få en enhetlig programmeringsbild i Visual Basic så kopieras detta HRESULT-värde till Err-objektet. Metoderna verkar istället returnera (om något) det värde som deklarerats som `[out, reval]` i IDL-filen (d.v.s. i COM-klassens *type library*). Metoden `Slumpa()` har definierats enligt följande i IDL-filen:

```
  [id(1), helpstring("method Slumpa")] HRESULT Slumpa([in] int max,
                                                      [out, retval] int* slumptal);
```

implementerats enligt följande i Visual C++:

```
  STDMETHODIMP CSlumpCpp::Slumpa(int max, int *slumptal) {  //Koden för metoden  }
```

anropas i Visual Basic enligt följande:

```
  Sub Command1_Click()
     On Error Goto Command1_Error        'Om fel gå till etiketten Command1_Error:
        'Deklarera variabler och skapa objekt
```

```
    intSlumptal = objSlump.Slumpa(11)   'Anropa metoden Slumpa() i komponenten
    MsgBox "Det gick att slumpa!"       'Visa meddelanderuta om att anrop lyckades

Exit Sub                                    'Avsluta medoden
Command1_Error:                             'Etiketten Command1_Error:
  MsgBox "Fel nummer: " & Err.Number & " uppstod i komponenten Slump."
End Sub
```

och i Visual C++ enligt följande:

```
HRESULT hr = pSlump->raw_Slumpa(max, &m_Slumptal);          //Anropa metod
if(SUCCESSED(hr))                                           //Om anrop lyckades
  MessageBox("Det gick att slumpa!");
```

Returvärdet från metoder i C++ kan testas med två fördefinerade makron: `FAILED()` och `SUCCEEDED()`. Dessa makron tar ett HRESULT-värde som parameter och returnerar sant eller falskt. Alla metodanrop till komponenter i C++ bör kontrolleras så att de lyckades innan vi fortsätter (vi kommer se ett annat sätt att hantera fel lite längre ner).

5.1.6 Innan avslut av applikation

Som nämnts ovan så ska man alltid anropa metoden `Release()` för varje gränssnittspekare man har till objekt och sen sist `CoUninitialize()`.

```
    pSlump->Release();  //Meddela komponenten att du är klar & att den kan förstöra.

    CoUninitialize();   //Meddela COM att du är klar med COM-miljön.
}
```

5.1.7 Applikationens kod i sin helhet

För att sätta allting på plats och visa hur felhantering med makrot FAILED() fungerar visas koden för applikationen i sin helhet här. I koden finns även kod som skriver ut ledtext och annat som inte visats ovan.

```
#include "stdafx.h"                         //Lades till av guiden
#include <iostream.h>                       //För att använda cout
#include "..\KompTestCpp\KompTestCpp.h"     //Inkludera header-filen för komponenten

int main(int argc, char* argv[])
{
   int i = 0;                       //Räknare för loop
   int max = 10;                    //Maxvärde för slumptal
   int slumptal = 0;                //För returvärden
   HRESULT hr;                      //För resultat från COM-metoder
   ISlumpCpp* pSlump = NULL;        //Pekare till gränssnitt

   hr = CoInitialize(NULL);         //Initialisera COM för att exekverar i en STA
   if(FAILED(hr))    //Om inte COM initierat, meddela och avsluta program
   {
      cout << "FEL: Det gick inte att initiera COM!" << endl;
      return 1;
   }

   hr = CoCreateInstance(__uuidof(SlumpCpp),    //GUID för COM-klass
                  NULL,                         //Icke aggregerat
                  CLSCTX_INPROC_SERVER,         //In-process komponent
                  __uuidof(ISlumpCpp),          //GUID för gränssnitt
                  reinterpret_cast<void**>(&pSlump));  //Variabel för pekare

   if(FAILED(hr))    //Om inte objekt skapades, meddela och avsluta program
```

```
      {
         cout << "FEL: Det gick inte skapa objektet!" << endl;
         return 1;
      }
         //Lite ledtext
      cout << "Detta program tar fram slumptal och skriver ut det fran en loop."
         << endl << "I varje loop pausar programmet." << endl;

         //Loopa 10 gånger och skriv ut slumptalet
      for(i = 0; i < 10; i++)
      {
         hr = pSlump->Slumpa(max, &slumptal);      //Anropa metod i komponent
         if(FAILED(hr))                            //Om anrop inte lyckades
         {
            cout << "FEL: Det gick inte att anropa metod!" << endl;
            return 1;
         }
         cout << "Slumptal: " << slumptal << endl; //Skriv ut slumptal
         Sleep(1000);                              //Pausa i 1000 milisekunder
          }

      pSlump->Release();//Meddela komponenten att du är klar och kan förstöras
      CoUninitialize(); //Går inte dessa metoder bra så ska programmet ändå avslutas
      return 0;         //Returnera sant då applikationen slutförts
   }
```

5.2 Ett enklare sätt att använda komponenter i Visual C++

Som har kunnat ses ovan så krävs det en hel del kunskap om COM för att använda komponenter i Visual C++. Lyckligtvis finns det ett enklare sätt att använda komponenter – genom att importera komponentens *type library* som finns i komponentens TLB- och/eller DLL-fil. Och för komponenter som vi inte utvecklar själva finns inte alltid *header*-filerna tillgängliga utan vi måste importera komponentens *type library*.

5.2.1 Direktivet #import

`#import`-direktivet placeras (oftast) längst upp efter alla `#include`-direktiv.

```
#include "stdafx.h"
#include <iostream>
#import ”C:\Komponent.dll”        //Importerar från DLL-fil
#import ”C:\AnnanKomponent.tlb”   //Importerar från TLB-fil
```

Något som detta direktiv gör är att automatiskt skapa skalklasser (*wrapper classes*) för komponentens metoder. Dessa skalklasser gör det möjligt för klienter i Visual C++ att bl.a. göra direkta anrop av metoder i objektet.[26] När kompilatorn finner direktivet `import` så genererar den två stycken filer – en TLH-fil[27] och en TLI-fil[28] som placeras i ”utkatalogen” för projektet (t.ex. DEBUG-katalogen). Dessa filer länkas in av kompilatorn vilket gör att koden i projektet får tillgång till definitionen av komponentens gränssnitt (precis som med direktivet `#include`).

För att slippa behöva skriva COM-serverns (komponentens projekts) namn framför varje referens till komponentens *type library* kan man lägga till direktivet `no_namespace` sist på raden med `#import`-direktiv.

[26] Se Eddon[99] för mer information om detta.
[27] *Type Library Header*
[28] *Type Library Implementation*

```
#import "KompEtt.dll"                              //Namn måste anges
#import "KomponentMedLangtNamn.tlb" no_namespace  //Behöver inte ange namn

main(){
   KompEtt::Komponent1* pKompEtt;  //Pekare till objekt från server KompEtt.dll
   Komponent2 pKompTva; //Pekare till objekt från server KomponentMedLangtNamn.tlb
   //Resten av koden
}
```

(En fördel med att börja med att skriva denna rad med kod är att IDE:n för Visual C++ sen kan använda s.k. IntelliSense för att visa på vilka metoder och egenskaper som finns tillgängliga i komponenten.)

5.2.2 Smarta pekare

En annan fördel man vinner med att använda direktivet `#import` är att man automatiskt får tillgång till s.k. smarta pekare för komponenten. Dessa smarta pekare finns för att göra COM-programmering lite lättare och säkrare. Man ska dock vara medveten om att det blir mer kod som måste exekveras, men det kan kanske vara ett pris man är beredd att betala för att få mer lättläst kod och förhoppningsvis mindre risk för fel.

För att både deklarera en gränssnittspekare och skapa en instans av komponent så använder man sig av konstruktor för komponentens skalklassen. Skalklassen för den smarta pekaren får samma namn som komponentens gränssnitt och med suffixet "`Ptr`". Till konstruktorn bifogar man COM-klassen för den komponent man vill instansiera.

```
IKomponentPtr pKomp(__uuidof(COMKlass));  //Skapa en instans av klasen COMKlass

ISlumpCppPrt pSlump(__uuidof(SlumpCpp));  //Skapa en instans av SlumpCpp
```

Genom att använda smarta pekare så kan man skriva koden på ett sätt som påminner om Visual Basics syntax. Bl.a. så kan man tilldela och hämta värden från komponents egenskaper på ett sätt som liknar Visual Basic eftersom fel hanteras på ett annat sätt. Om vi skriver om exemplet från 5.1.4 så skulle det kunna se ut enligt följande.

```
   pSlump->MaxVarde = max          //Tilldelning till egenskap som i VB
   pSlump->InitieraSlumptal()      //Initiera slumptalsgenerator – bifoga maxtal
   for(i = 0; i < 10; i++)
   {
      slumptal = pSlump->Varde();  //Tilldelning av variabel från komp. som i VB
      cout << "Slumptal: " << slumptal << endl;  //Skriv ut slumptal
   }
```

5.2.3 Reviderad version av applikationen

Som vi kommer se från koden nedan så blir koden lite mer lättläst och vi behöver inte anropa COM-funktioner som `CreateInstance()` med alla dess parametrar och `Release()` – det gör den smarta pekaren åt oss. Som nämnts ovan så finns en möjlighet till rikare felhantering – vi behöver inte nöja oss med bara ett HRESULT-värde utan kan använda *exceptions* i en `try{...}catch{...}`-sats.

```
#include "stdafx.h"
#include <iostream.h>
#import "..\KompTestCpp\Debug\KompTestCpp.dll" no_namespace
```

```
int main(int argc, char* argv[])
{
   int i = 0;                    //Räknare för loop
   int max = 11;                 //Maxvärde för slumptal (=max+1)
   int slumptal = 0;             //Variabel för att hantera returvärdet.

   CoInitialize(NULL);                        //Initialisera COM-miljön
   try //Om fel...
   {
      ISlumpCppPtr pSlump(__uuidof(SlumpCpp));  //Deklarera och skapa komponent
                                 //OBS! Måste ske efter CoInitialize()!
         //Ledtext
      cout << "Detta program tar fram slumptal och skriver ut det fran en loop."
         << " I varje loop pausar programmet." << endl;

         //Loopa 10 gånger och skriv ut slumptalet
      for(i = 0; i < 10; i++)
      {
         slumptal = pSlump->Slumpa(max);
         cout << "Slumptal: " << slumptal << endl;
         Sleep(1000);
      }
         //Anropet Release behövs inte då man använder smarta pekare
      pSlump = NULL; //Bör dock tala om att man är klar med objektet (som i VB)
   }
   catch(const _com_error& Err)  //... meddela användare
   {
      TCHAR text[255] = {0};                        //Deklarera en strängvariabel
      wsprintf(text, _T("Error: 0x%x "), Err.Error());  //Bygg felsträng
      cerr << text << endl;              //Skriv ut felmeddelande till fel-stream
   }
   CoUninitialize(); //Meddela COM att du är klar med COM-miljön.

   return 0;
}
```

5.2.3.1 Ett litet PS

Ibland händer det att man får ett meddelande (motsvarande det nedan) när man kompilerar klientprogram med import-direktivet. Det är bara att kompilera en gång till så brukar det fungera…

```
fatal error C1083: Cannot open compiler generated file:
'c:\student\komptestconsole\debug\KompTestCpp.tlh': Permission denied
```