Serviceorienterad arkitektur – en översikt

Som vi tidigare berättat kommer vår arkitektwebb under kvartalet oktober – december 2004 att domineras av temat serviceorienterad arkitektur, ofta förkortat som SOA. Under var och en av kvartalets månader kommer vi att publicera en artikel som handlar om det som är kvartalets tema. Det här är oktober månads artikel, och den ger en översikt över begreppet SOA. Författare till artikeln är Sten Sundblad på Sundblad & Sundblad i Uppsala.

Lite historik skadar inte

Jag tänkte börja med lite historik. SOA har inte exploderat fram; det är inte heller frågan om någon revolution som bryter med allt det gamla. SOA är i stället ett naturligt nästa steg i en utveckling som har pågått sedan datorernas barndom. Det mesta av det du kan idag har du stor nytta av när du arbetar i en serviceorienterad miljö.

Det finns inte heller något *motsatsförhållande* mellan objektorientering och serviceorientering. I en serviceorienterad miljö finns det massor av objekt. Du kan i och för sig hoppa över objektorienteringen och bygga insidan av dina services på ett äldre sätt, men i normalfallet är det objektorienteringen som dominerar *insidan* av en service. Det enda som skiljer en service från ett objektorienterat program är det sätt på vilket en service respektive ett ”vanligt” program exponerar sin funktionalitet.

Funktionsorienterad programmering

Jag tänker inte gå så långt tillbaka som till datorernas barndom, som inträffade någon gång i mitten eller slutet av 1940-talet. (Egentligen borde jag ha skrivit datamaskinernas barndom, för det var så de kallades ända in på 60-talet då begreppet dator etablerades. Visste du att den som stiftade benämningen dator utgick från begreppet traktor, något som många reagerade mot. Många vägrade acceptera den nya benämningen och fortsatte att kalla datorerna för datamaskiner; begreppet dator hade inte ens en avlägsen chans att slå igenom, menade man.)

Nej, det räcker att gå tillbaka till de funktionsorienterade program som skrevs i något tämligen strukturerat programmeringsspråk som Cobol, Pascal, PL/1, C eller, speciellt när det gällde de så kallade mikrodatorerna, någon sen variant av BASIC. I det sammanhanget bör man kanske också nämna dBASE, som många bedömde med tiden skulle bli det dominerande programmeringsspråket för affärstillämpningar.

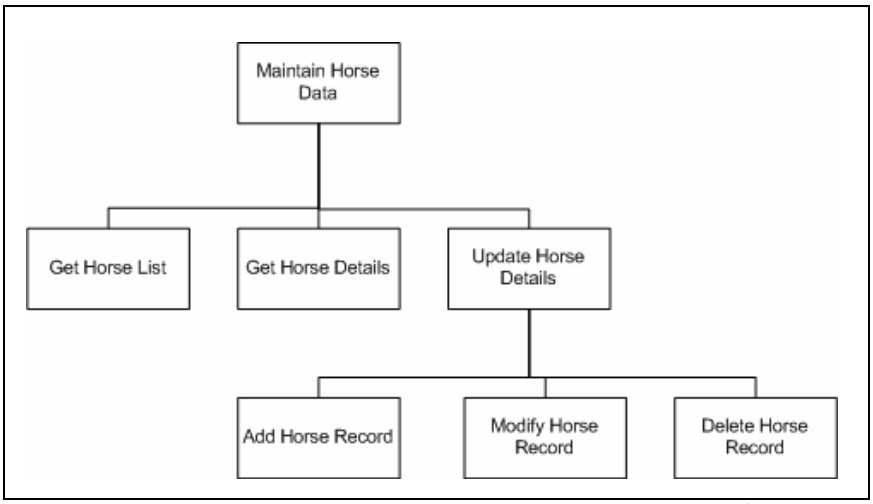
Det som var karaktäristiskt för funktionsorienterade program var möjligheten att utforma och skriva funktioner och procedurer som ersatte de gamla subrutinerna. Man började tala om ”goto-less programming”. Det här var ett stort framsteg jämfört med de tidigare mera ostrukturerade program man kunde bygga med programmeringsspråk som inte hade stöd för funktioner och procedurer. Dessa baserades på numera så gott som okända kommandon som GoTo och GoSub. Dessa kommandon användes till att ”hoppa” från ett ställe i ett program till ett annat.

För första gången blev det lönsamt att lägga en designaktivitet framför programmeringsinsatsen. En designer kunde dela in ett tänkt program i huvudrutiner (Main) och underrutiner som procedurer och funktioner. Programmerarens uppgift blev att implementera en i förväg genomtänkt design. Figur 1 är ett exempel på en typ av diagram som ofta användes under den här tiden.

Det är under den här tiden många av de principer för strukturering av program som vi fortfarande arbetar efter kom till. Det var ändå inte så mycket frågan om nytänkande utan om utnyttjande av sedan länge kända strategier för problemlösning. Några av de viktigaste av dessa strategier, de som ansågs vara de beståndsdelar som tillsammans utgjorde den ursprungliga filosofin om strukturerad programmering, följer här:

- *Principen om abstraktion* innebär att man fokuserar på de aspekter som är viktiga från en viss utgångspunkt och bortser från alla andra aspekter. Syftet är att representera problemet eller lösningen i förenklad form och därmed tydliggöra de viktiga aspekterna och undertrycka de mindre viktiga.

- *Principen om formalitet* innebär att man följer ett rigoröst och metodiskt angreppssätt vid lösning av problem.
- *Härska genom splittring* innebär att man delar upp ett stort och komplext problem i en uppsättning mindre och enklare problem, som vart och ett är lättare att förstå och lättare att lösa.
- *Hierarkisk organisation* innebär att man organiserar en lösnings alla komponenter i en trädliknande hierarkisk struktur. Nedbrytningen gör att lösningen kommer under bättre kontroll och kan byggas nivå för nivå med högre detaljeringsgrad för varje nivå. Vi refererar åter till Figur 1 som ger ett exempel på hierarkisk organisation av ett program och dess funktioner.



Figur 1 - Diagram som visar den hierarkiska organisationen av ett program som underhåller ett hästregister. Denna typ av diagram var vanlig under "strukturrevolutionen".

Objektorienterad programmering

Bland andra principer för vad som kom att kallas för *software engineering* fanns principen om dold information (information hiding), principlen om placering (localization) och principen om konceptuell integritet (conceptual integrity). Dessa principer var förmodligen viktigare än något annat för framväxandet av objektorientering.

Ett av problemen med funktionsorienterad programmering var att flera funktioner och procedurer kunde dela på samma data. Det här gjorde funktionsorienterade program mindre stabila än de borde vara, speciellt över tiden. En ändring av en funktion kunde påverka en annan funktion i en helt annan del av programmet, nämligen om båda delade på användningen av samma data. Den första funktionen kunde på ett för den andra funktionen oväntat sätt ändra data som den andra funktionen var beroende av. Detta problem kom att kallas för dominoeffekten; om en bricka föll kunde alla falla i en kedja av effekter.

Lösningen kom att kallas för objektorientering. I stället för att låta flera funktioner dela på samma data kapslar objektorienteringen in data och de funktioner som arbetade med detta data i något som kom att kallas objekt. Objekt kunde struktureras så att ingen utanför objektet kom åt dess data; enda sättet att komma åt ett objekts data blev att anropa en av objektets publicerade metoder. I och med att två eller flera funktioner därmed inte kom åt att använda samma data undanröjdes risken för dominoeffekt. Det här är en av de viktiga orsakerna till att objektorienterade program generellt sett är stabilare än funktionsorienterade.

Det är alltså *principen om dold information* som gör att externa enheter inte kommer åt ett objekts data; det är ju gömt för alla utomstående. Det är *principen om placering* som säger att data skall placeras inne i de objekt som har huvudansvaret för detta data. Det är *principen om konceptuell integrite*t som gör att användare av ett objektorienterat program kan lita på att dessa principer är allmänt förekommande i programmet.

Det är särskilt tre egenskaper som anses vara utmärkande för objektorienteringen och för objektorinterade program. De är:

- *Inkapsling* innebär det vi nyss har avhandlat. Objekt kapslar in data (även kallat tillstånd) och beteende i en sammanhållen enhet. Endast det tillstånd och det beteende ett objekt vill exponera blir åtkomligt utanför objektets gränser. Därmed blir också gränser, och tvingande respekt för gränser, en påfallande egenskap inom objektorienteringen.
- *Arv* är en mekanism för återanvändning. Arv delar in objekt i supertyper och subtyper. Det tillstånd och det beteende som placerats i en supertyp (eller basklass) kan ärvas av typens subtyper (eller arvingar). Arv erbjuder kantlös återanvändning eftersom arvingen kan använda basklassens tillstånd och metoder som vore de arvingens egna. Endast det som basklassen deklarerat som privat blir basklassens ensak i förhållande till klassens alla arvingar.

  En viktig praktisk restriktion är att ett objekts arvingar verkligen bör vara naturliga subtyper till basklassen. Vanligast förekommande, i varje fall i resonemang om arv, är att ett objekt av entitetstyp ärver av ett entitetsobjekt av en mer generell klass. Entitetsobjekt av klassen *RaceHorse* kan till exempel ärva av generaliseringen *Horse*. Mindre vanligt i resonemang om arv är andra typer av objekt, sådana som främst representerar beteende. Ändå är hela .NET-miljön i princip uppbyggd av sådana objekt, något som tydligt framgår av referensdokumentationen av .NET Framework. Varje klass i .NET Framework har i denna dokumentation hela sin arvshierarki dokumenterad.
- *Polymorphism* är ett begrepp som även existerar utanför området objektorienterad programmering. Det betyder egentligen förmåga att uppträda i många former. Inom objektorientering representerar begreppet möjligheten att i en klass som ärver från en annan klass definiera ett från supertypen avvikande beteende för en eller flera metoder.

  Klassen anställd kan till exempel ha olika arvingar, där varje arvinge representerar en kategori av anställda. För vissa kategorier kan det finnas ett eget och från normen avvikande sätt att räkna ut den anställdes lön. Polymorphism gör det möjligt för de olika kategorierna att visa upp ett gemensamt ansikte (läs gränsnitt) för löneuträkning, men att ändå räkna ut lönen på passande sätt för respektive kategori. Poängen är att klienten inte behöver veta något om vilken kategori av anställd den begär löneuträkning för; objektet själv vet vilken kategori det tillhör och kommer att välja sin egen specifika version av löneuträkningsmetoden.

  Återigen är *det principerna om hiearkisk ordning och dold information* som regerar. Alla objekt som ingår i arvshierarkin visar upp ett gemensamt ansikte med identisk metodsignatur, men de kan internt och dolt för klienten bete sig på olika sätt.

Objektorienterad programmering har gjort underverk för programkvaliteten, men blev ändå inte en omedelbar succé. De första objektorienterade programmeringsspråken genererade program som i bästa fall var långsamma och i många fall var plågsamt långsamma. Det var inte ovanligt att programägare uttryckte att de ”aldrig skulle tillåta sina program att bli objektorienterade”. Programmen var alltför viktiga för det, menade man.

Allt det här är borta nu. Objektorienterade program är normen, allt annat är undantag. Detta gäller i varje fall för alla de program som skrivs nu – äldre program är naturligtvis fortfarande funktionsorienterade. Leverantörerna av objektorienterade miljöer har gjort underverk, och det är idag ovanligt att någon av prestandaskäl avstår från objektorientering. Snart kommer vi faktiskt till och med att ha det första icke-triviala operativsystemet som är skrivet med objektorienterad teknik. Det är nästa version av Windows jag då tänker på.

Distribuerade objekt

Den tekniska och marknadsmässiga utvecklingen av nätverk har förändrat villkoren för programutveckling. Det började med den så kallade client/server-modellen, där den tidigare monolitiska tillämpningen delades upp i två delar. Den ena delen – klientdelen – frikopplades från den andra delen – serverdelen – och vi fick två närmast jämbördiga tillämpningar som talade med varandra över datorgränser. Eftersom serverdelen i varje fall i princip var en SQL-databas räckte det med möjligheten att sända SQL-satser över nätverket för att få modellen att fungera – det fanns då inget behov av objektorienterad kommunikation mellan datorer.

Ett problem med client/server-modellen rörde placeringen av det regelverk som skall kontrollera ny eller ändrad information innan den persistent lagras. Om regelverket placerades i servern måste det

programmeras inne i databasen, som lagrade procedurer och triggers, med någon variant av SQL. Det är inte särskilt praktiskt, eftersom SQL inte är tänkt för procedurell programmering, eftersom det inte finns en uppsjö på kvalificerade SQL-programmerare, och eftersom det också är svårt att avlusa och versionshantera SQL-objekt. Svårigheten och komplexiteten växer med storleken på SQL-objekten; vi har sett exempel på lagrade procedurer som i utskrivet skick kräver så mycket som 18 A4-sidor. (Det var det värsta exempel vi har sett, och vi vill gärna betona två saker. Det var inte vi som hade skrivit den☺. Och vi behöver väl inte ens säga det: den fungerade inte.)

Alternativet att placera regelverket i klienten löste dessa problem, eftersom klienten helt naturligt är programmerad i språk som är lättillgängliga, som medger enkel och effektiv avlusning och versionshantering. Problemet blev ett annat. Hur skulle man kunna garantera att alla transaktioner som förändrade databasen verkligen gick igenom regelverket, och att de klientprogram som databasen skulle skyddas mot också innehöll regelverket? Och hur skulle man över tiden kunna garantera att alla klientprogram innehöll en korrekt och komplett version av regelverket och dessutom slussade alla transaktioner genom det?

Lösningen blev de tre- och flerskiktade exempel på arkitektur som växte fram i skuggan av distribuerade objekt. Utanför Microsoftvärlden blev CORBA den dominerande teknologin för distribuerade objekt; inom Microsoftvärlden blev det först DCOM och sedan .NET Remoting.

Alla dessa miljöer löste client/server-modellens problem. Regelverket kunde placeras centralt i särskilda applikationsdatorer, där det kunde programmeras i språk som tillät avlusning och versionshantering och där kunskap om programmering i språket var spridd. Alla klienttransaktioner kunde med hjälp av databasens behörighetssystem tvingas igenom applikationsskiktet och därmed regelverket. Endast ett problem återstod, men det visade sig vara ett stort och viktigt problem.

I princip krävde den nya modellen ett stort samförstånd mellan klient och server. Båda måste äga kunskap om den andres programmeringsspråk, objektmodell och ofta även applikationsserver. Distribuerade objekt kräver egentligen en homogen exekveringsmiljö, men en sådan är mer undantag än regel. Det finns knappast ett enda medelstort eller stort företag i hela världen som har en homogen IT-miljö, och frågan är om det finns särskilt många små företag som har det.

Behovet att integrera program, exekverande i olika miljöer, visade sig vara stort. Många företag har under ett antal år kunnat leva riktigt gott på det, eftersom de tog fram oerhört användbara och rejält behövda men också kraftigt investeringskrävande programprodukter för integrering av program i heterogena miljöer.

Don Box, som är en av världens absolut främsta experter på Microsofts teknologier för distribuerade objekt, har sagt att ”vi trodde att det skulle gå, och vi sträckte ut teknologierna för distribuerade objekt så långt vi någonsin kunde, men det visade sig att det inte fungerade. Vi behövde något annat!” Don Box och många andra menar att detta ”något annat” är distribuerade services. Mycket talar för att det är precis på det sättet det ligger till.

Distribuerade services

Funktionsorienterade programmeringsspråk löste flera problem som fanns med de tidigare ostrukturerade språken. En viktig egenskap hos de nya språken var att de tillät oss att studera problem och lösningar på en högre abstraktionsnivå. Detta ledde till nya framsteg som i sin tur ledde till andra och nya problem på en högre nivå än de gamla. Objektteknik hjälpte oss att lösa dessa problem och skjuta fram positionerna ytterligare. Som ett brev på posten fick vi integrationsproblemen, och det är dem vi måste försöka lösa nu. Om vi gör det, och det är vi säkert på väg att göra, kommer vi att kunna skjuta fram positionerna ytterligare och bygga kommande system utifrån en ny arkitektur.

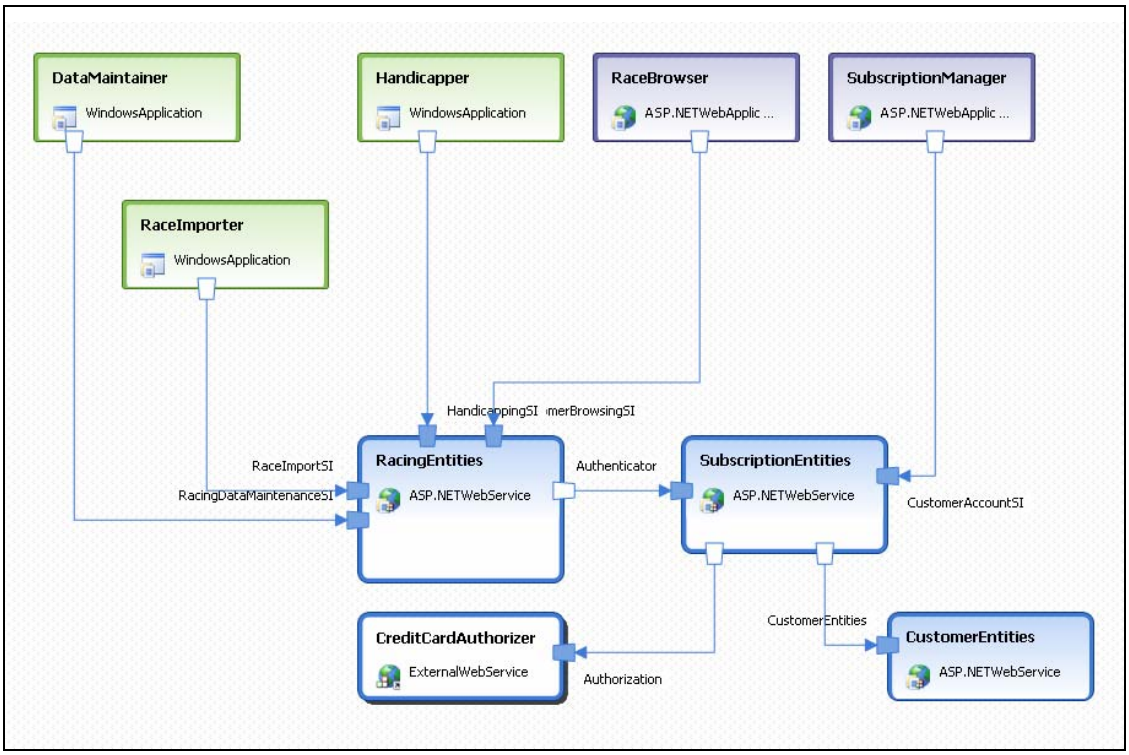
Lösningen på många integrationsproblem kommer att ligga i användningen av XML och SOAP. Som varje läsare av den här artikeln säkert vet är XML en formell standard för formatering av data och SOAP ett XML-baserat protokoll för formatering av den strukturerade och typade information som utbyts mellan jämlikar (peers) i en decentraliserad och distribuerad exekveringsmiljö. I översiktsavsnittet av den dokumentation som beskriver SOAP-standarden framhålls att ett SOAP-meddelande formellt sett är specificerat som ett XML infoset som ger en abstrakt beskrivning av meddelandets innehåll. Typiskt fraktas

ett SOAP-meddelande över HTTP, men även andra transportprotokoll som TCP kan vara bärare av SOAP-meddelanden.

Kombinationen av SOAP och XML har givit upphov till fenomenet Web Services. En web service är ett program som exponerar ett SOAP-baserat meddelandeorienterat programmeringsgränssnitt. Enda sättet att kommunicera med en web service är att förpacka ett XML-baserat meddelande i ett så kallat SOAP-kuvert. Även ett eventuellt svar från en web service förpackas på detta sätt.

SOAP är XML-baserat, och ett SOAP-meddelande är förpackat som ett XML-dokument eller en XML-ström. XML är helt och hållet språk- och plattformsoberoende. Därför kan SOAP och XML tillsammans lösa problemet med integration av heterogena IT-miljöer.

Figur 2 är ett diagram som visar hur tre Windowsapplikationer och två webbapplikationer konsumerar tjänster från tre web interna web services. Alla dessa applikationer och services är deklarerade som Microsoftkomponenter. Det finns emellertid ytterligare en komponent i diagrammet. Det är en extern web service, kallad CreditCardAuthorizer, som kan hjälpa SubscriptionEntities med att auktorisera kreditkortsköp. Denna service är extern, och vi har ingen aning om vilken miljö den exekverar i, vilket programmeringsspråk den är skriven i eller vilken databashanterare den använder. Det behöver vi inte heller veta. Det enda vi behöver veta är att den är en web service, utformad för interoperabilitet, och det kan vi avgöra efter att ha studerat den WSDL-fil i vilken den exponerar sitt gränssnitt till omvärlden.



Figur 2 - Diagram som visar hur olika klienter och klienttyper konsumerar web services

Serviceorienterad arkitektur

Så har vi då äntligen kommit fram till ämnet för artikeln – serviceorienterad arkitektur eller SOA.

SOA bygger på samma idéer som SOAP och XML men är inte beroende av någondera. SOA-idén är helt enkelt att organisera IT-system som uppsättningar av services (eller tjänster) som kan nås via meddelandebaserade gränssnitt. Idén föddes i CORBA-kretsar för mer än 10 år sedan, men det är först tack vare XML, SOAP och web services som den blivit riktigt intressant. Skälet är naturligtvis att det är dessa företeelser som gjort det möjligt för en service att bli språk- och plattformsoberoende.

När man i beskrivande ordalag talar om serviceorienterad arkitektur brukar man bland annat räkna upp följande egenskaper som en service bör besitta:

- Den bör vara autonom.
- Den bör ha ett meddelandebaserat gränssnitt.
- Den bör ha egen logik som kapslar in och skyddar sitt eget data.
- Den bör vara tydligt avgränsade från och löst kopplad till omvärlden.
- Den bör dela schema med sina konsumenter – inte klass.

Det finns tydliga överlappningar mellan dessa egenskaper. Låt oss ändå kort säga något om var och en av dem.

Att vara autonom

Att vara autonom är en av de kanske viktigaste egenskaperna hos en service. Det innebär bland annat att den skall kunna utvecklas, installeras, underhållas och vidareutvecklas oberoende av sina konsumenter.

Att vara autonom innebär också att ta fullt ansvar för den egna säkerheten och i princip inte lita på att någon konsument är behörig att ta del av tjänsten. Det innebär också att vara tämligen misstänksam för att skydda sig mot obehörig och illvillig påverkan utifrån.

Meddelandebaserade gränssnitt

Vi har redan nämnt att ett meddelandebaserat gränssnitt är centralt för en service. Gränssnittet behöver inte nödvändigtvis ha formen av en web service utan kan också vara baserat på meddelandeköer. Ändå är det ganska säkert web services som kommer att dominera som servicegränssnitt, eftersom även allmänt förekommande köprodukter, som IBM's MQ Series, har proprietära drag och inte är helt plattformsoberoende.

Idén är att ingen konsument av de tjänster som en service exponerar skall komma i kontakt med någon inre del av denna service. Enda sättet att åtnjuta tjänsterna är att sända ett meddelande som ”ödmjukt ber att få den här tjänsten utförd”. Meddelandet tas om hand av tjänstens logik som tar ställning till om uppdraget ska utföras och som i så fall också utför det.

En viktig restriktion för vidareutveckling av en service är att inget gränssnitt som redan har exponerats får ändras på ett sådant sätt att tjänstens konsumenter påverkas av ändringen. Här skiljer sig inte serviceorienteringen från objektorienteringen, i varje fall inte principiellt. Praktiskt sett finns det ändå två viktiga skillnader:

- Det är ännu viktigare för en service att i all framtid fortsätta att stödja publicerade gränssnitt. Det beror i så fall på att en service potentiellt, genom sitt plattformsoberoende och sin eventuella exponering över Internet, riskerar att ha mer spridda konsumenter utanför serviceägarens kontrollsfär än vad som är vanligt för objekt.
- En servicemetod kan kapsla in både anropsparametrar och svar i var sitt dokument. I stället för att sända till exempel två integer-variabler kan du kapsla in dem i ett XML-dokument och sända detta dokument som enda anropsparameter. När du anropar en service sänder du då ett dokument till den och får ett annat tillbaka. Skulle du behöva lägga till en parameter för nytillkomna behov ändrar du inte servicemetodens signatur, bara innehållet i dokumentet.

Inkapsling av data

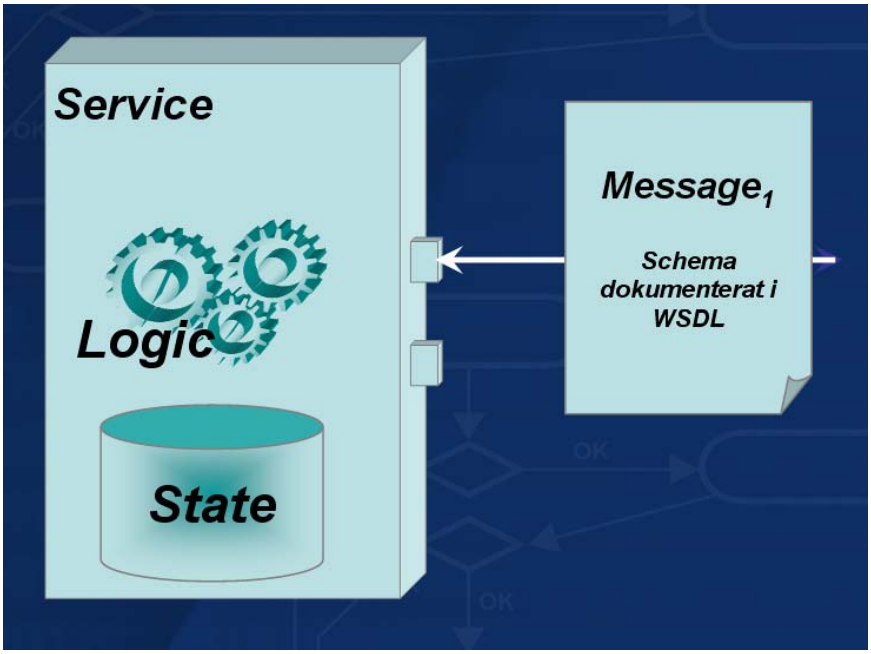
En service kapslar in allt sitt data och gör det inte direkt tillgängligt för någon enda konsument. Det enda sättet för en konsument att komma åt tjänstens data är att sända ett meddelande till den och be att få ett uppdrag utfört. Meddelandet når aldrig datat, alltid bara den logik tjänsten använder för att kapsla in sitt data.

Man brukar tala om att datat ligger inlåst i ett säkert kassaskåp (en ”strongbox”). Endast kassören, som utgörs av tjänstens uppsättning av dataåtkomstklasser, har tillgång till nyckeln.

Även objekt kapslar in data; inkapsling av data är ju en av de bärande principerna för objektorientering. Det finns emellertid en stor skillnad:

- Ett objekt kapslar in flyktig data som lever i datorns primärminne. För persistent lagring är objektet beroende av någon extern komponent som delas med andra objekt.
- En service består av ett antal objekt, som vart och ett kan kapsla in objekttillstånd i primärminnet men kapslar till skillnad från objekt *också* in det persistenta data som typiskt lever i en databas och som föder varje enskilt objekt med behövligt tillstånd.

Figur 3 ger en översiktsbild av en service och visar hur den internt består av logik och tillstånd, men också att dess enda kontakter med omvärlden sker via ett meddelandeorienterat gränssnitt. Enda sättet att komma i åtnjutande av dess tjänster är att sända den ett meddelande.



Figur 3 - En service kapslar in sitt tillstånd i logik och erbjuder ett meddelandeorienterat gränssnitt till sina konsumenter.

Tydlig avgränsning från och löst kopplad till omvärlden

En av de bärande principerna för serviceorientering är den tydliga avgränsningen mellan en service och dess omvärld. Ett uttryck för denna princip är de lösa och kortvariga förbindelserna mellan konsument och service. Med objekt är det annorlunda eftersom objektorienteringen låter konsumenter kopplar upp sig till objekt på ett fast och stadigvarande sätt.

Förbindelsen mellan konsument och objekt kan vara långvarig; den förstörs inte automatiskt när ett uppdrag har fullföljts. Den kräver också att konsument och objekt är kompatibla med varandra och delar på sådant som objektmodell, programmeringsspråk och kanske också applikationsserver.

Förbindelsen mellan konsument och service är alltid kortvarig och förstörs när en service har svarat på ett mottaget meddelande. Behöver konsumenten sända ytterligare ett meddelande sätts alltid en ny förbindelse upp. Det finns inte heller något krav på att konsument och service skall dela på något annat än meddelandets struktur och det kontrakt som styr kommunikationen mellan konsument och service.

Delar schema – inte klass

En viktig princip för SOA är att konsument och service har en gemensam uppfattning om de scheman som bestämmer meddelandens struktur. När du formger en service bör du utgå från att de konsumenter som kommer att använda denna service kan ha en helt annan objektmodell än den du utgår från för din service.

Därför skall en service inte publicera sin objektmodell, eller delar av den, utan endast de scheman som reglerar strukturen av de meddelanden som utbyts.

En annan viktig och näraliggande princip är principen om den svarta lådan. Inte heller den är ny – den kommer till och med från tiden före objektorienteringen. Den innebär att du först bör formge det eller de meddelandegränssnitt din service kommer att exponera och först *därefter* börja fundera på hur du skall formge interiören. Denna princip innebär till exempel att alla de XML-scheman och kontrakt som reglerar kommunikationen mellan konsument och service kommer att existera innan de objekt som implementerar gränssnitten ens är påtänkta.

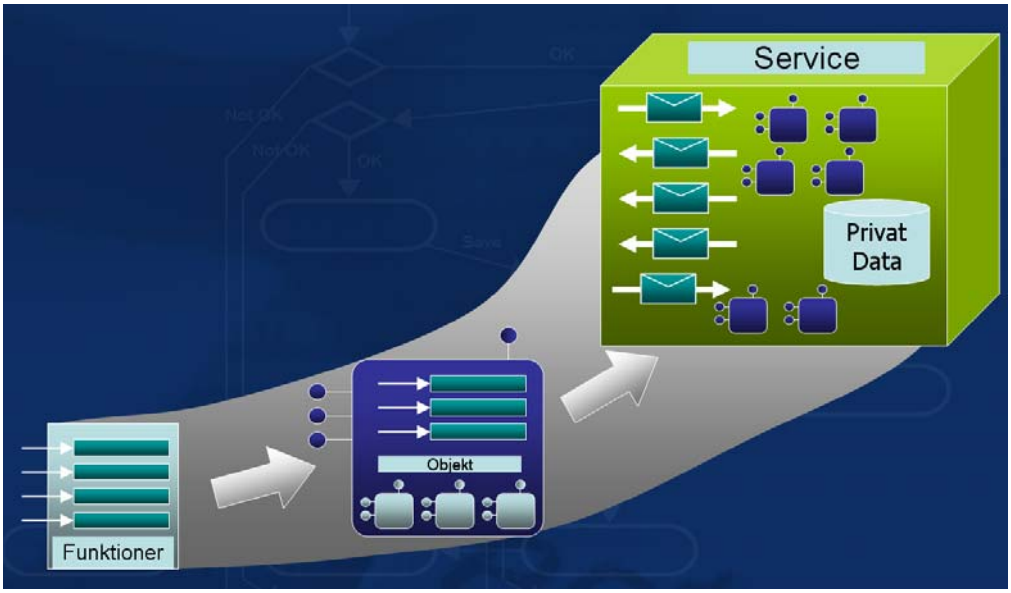
Objekt kontra services

Det finns alltså både skillnader och likheter mellan objekt och services. Det intressanta är emellertid att se hur de båda förhåller sig till varandra.

Det vore fel att säga att serviceorientering *förutsätter* objektorientering, för så är det inte. Man kan mycket väl utforma insidan av en service utan att använda objekt. I praktiken är det emellertid så gott som alltid så att objektorienteringen dominerar insidan av så gott som varje service.

Där objekt kapslar in tillstånd och beteende för ett enda verksamhetsobjekt, som till exempel en kund, kapslar en service in tillstånd och beteende för ett helt verksamhets- eller informationsområde. En kundservice kan alltså innehålla alla de objekt som beskriver en kund, alla de objekt som beskriver kundens engagemang och alla de objekt som beskriver de kontakter företaget har haft med kunden. En sådan service kapslar också in den persistenta lagringen av den information som beskriver alla dessa objekt.

Figur 4 är tänkt att visa ett par intressanta saker. Den första är att övergången till serviceorientering är ett naturligt nästa steg på den utveckling som inleddes när vi gick från strukturlös programmering till strukturerad och funktionsorienterad. Den andra är att övergången till serviceorientering leder till en högre ordning, där objekt och komponenter blir en del av denna högre ordning. Objekt och komponenter ingår alltså, tillsammans med privat data, som *delar* av en service i stället för att själva utgöra en installerbar helhet.



Figur 4 - Vi har gått från en funktionsorienterad till en objektorienterad paradigm. Nu är vi på väg att ta nästa steg mot serviceorientering.

På samma sätt som objektorienteringen höjde abstraktionsnivån jämfört med den äldre funktionsorienteringen höjer serviceorienteringen abstraktionsnivån jämfört med objektorienteringen. Det är alltså inget dramatiskt som sker när vi går över från en objektdominerad till en servicedominerad värld. Det är helt enkelt ett naturligt nästa steg i en pågående utveckling som dessutom kommer att fortsätta.

Kommande artiklar

Det kommer två artiklar till på temat SOA. Den första räknar vi med att publicera i november, och den har arbetsnamnet *SOA och data*. Den andra tänker vi publicera i december under arbetsnamnet *SOA och processer*.