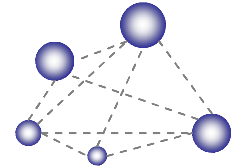


Vägen till tjänstebaserad arkitektur

Många är de företag som avslutat sina första piloter med Web Service och då kommit fram till att det kan tillföra mervärde för både IT och verksamhet. Nu är det dags att ta ett kliv framåt och se de nya möjligheterna med tjänstebaserad arkitektur. En samverkan mellan löst kopplade byggklossar som erbjuder funktionalitet och skapar den flexibilitet som krävs för en verksamhet i förändring.

Sammanfattning

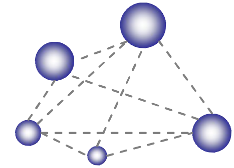
I det här dokumentet definieras SOA (Service Oriented Architecture) som ett paradigmskifte från monolitiska applikationer till små löst kopplade och samverkande tjänster. Här beskrivs en historik över hur annorlunda SOA har tolkats av affärsutvecklare kontra systemutvecklare men samtidigt också hur väl målen med SOA överensstämmer mellan varandras definitioner. Tjänstebegreppet i SOA har de senaste åren vävts ihop med processdriven utveckling som länge väntat på att få en högre status och bli en första klassens medborgare inom systemutvecklingen. Därför tar vi en titt på hur processdriven (arbetsflöden) utveckling är en naturlig del av SOA.

Tjänster ur ett bekant perspektiv

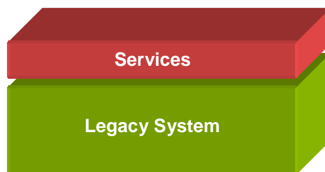
Web Services är vi ju bekanta med. Vi vet att Web Services kan användas på olika sätt, som exempelvis för att kommunicera mellan olika plattformar, genom brandväggar, på ett säkert sätt mellan två parter, för att överföra meddelanden eller att förlänga åtkomsten till komponenter ut på Internet. Vi vet även att en Web Service kommunicerar över ett antal olika protokoll och bygger på en standard som vi känner som SOAP. Möjligheten att skapa Web Services inkluderas nu i snabb takt i de flesta applikationsservrar och på alla plattformar. Man kan även se leverantörer som bygger ut sina API:er med WS-stöd för att göra sina produkter integrerbara.

Web Services har blivit den bilden en utvecklare har av tjänstebegreppet. Här följer tre stycken vanliga uppfattningar om vad SOA innebär:

En vanlig uppfattning är att SOA har med systemintegration att göra. Genom att man bygger ett tjänstlager (Web Service Facade Pattern) för att kunna öppna upp möjligheten att bygga applikationer som utnyttjar funktionaliteten i stordatorsystem och andra mer isolerade miljöer på ett standardiserat sätt. Den meningen delas oftast av dom som ser

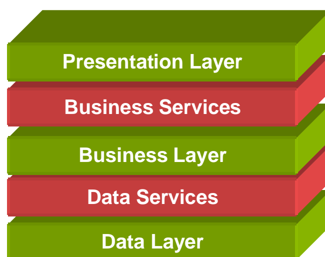


att stordatorsystemen kan abstraheras bakom en fasad av standardiserad åtkomst istället för att byggas om från exempelvis Cobol till Java, som också skulle öka integrationsmöjligheterna. Detta för att sedan, när fasaden står fast, kunna byta ut stordatorsystemen till modernare plattformar.



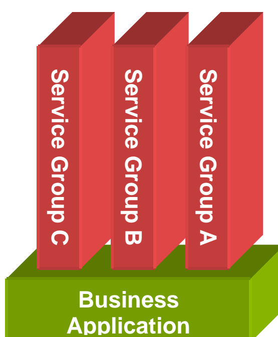
Figur 1 Web Service som fasad till stordatorer eller andra isolerade system.

En annan åsikt är att om man skjuter in ett eller flera tjänstelager i sitt system så får man SOA. Det hörs oftast från utvecklarleden som vill använda Web Service i sina applikationer. Genom att abstrahera ett lager med tjänster skapar de lösare koppling till underliggande lager och möjligheterna till mobilitet ökar.

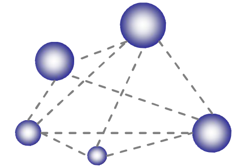


Figur 2 Web Service mellan lager i arkitekturen.

Man kan även höra att SOA är grupperingar av tjänster som add-on:s till applikationer för att skapa flexibilitet och integrationsmöjligheter. Den bilden får man oftast höra från produktutvecklare som har en stor applikation och vill öppna upp den för SOA-liknande tjänster.



Figur 3 Web Services som add-on på stora monolitiska applikationer.



Dessa uppfattningar är inte på något sätt felaktiga och de kan mycket väl spela en roll i utvecklingen av SOA. Men SOA är ett sluttillstånd och inte något man kan addera till den existerande systemmiljön genom fasader. SOA tränger in bakom fasaden och förändrar hela organisationen av applikationer och system.

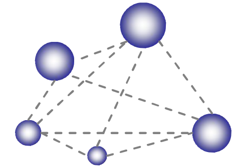
Visioner inom systemutveckling

Man har alltid strävat efter lägre kostnader och flexibilitet inom mjukvaruindustrin. Det ställer krav på återanvändning och utbytbarhet av mjukvara. Återanvändning bygger på möjligheten att återanvända existerade och redan skrivna moduler istället för att implementera dom igen, vilket resulterar i lägre kostnader. Flexibilitet är mjukvarans förmåga att vara föränderlig i tiden. En modul ska kunna bytas ut mot en annan för att spegla förändringar i den kontext där den används.

Historiskt sett har den ena modellen avlöst den andra för att möjliggöra återanvändning. Exempel på det är DLL, COM och .NET från Microsoft som alla strävat efter att möjliggöra återanvändning. Dynamic Linked Libraries (DLL) innebar att moduler kunde distribueras i annan form än som kod. Component Object Model (COM) är en disciplin som adderar lösare koppling (loosely coupled) mellan DLL:en och klienten genom Interface Based Programming. I och med .NET togs ett steg från löst kopplade komponenter till återanvändning baserat på allestädes närvarande protokoll och kod. Suns Java är ett annat bra exempel på samma modell som Microsofts .NET, eftersom det skapades för de flesta plattformar som exempelvis Unix, Linux, Windows och AS/400 för att på så sätt skapa en singulär dimension av återanvändning.

Objektorientering

Objektorientering blev mycket populärt eftersom det i språket gav stöd för just återanvändning. Vad man senare upptäckte var problemet med flexibiliteten i den mjukvara som var byggd med objektorienterade språk som C++ och Java. Implementationsarv kopplar moduler starkt till varandra vilket gör att återanvändning skedde på bekostnad av lägre utbytbarhet. Objektorienterade språk har i en tid efter det strävat efter att just bli flexiblare och det kan man se spår av i bland annat Aspect Oriented Programmering (AOP) som bygger på att modularisera "vanlig" funktionalitet (aspekter) genom injektion av funktionalitet vid exekvering i den så hårt kopplade arvsstrukturen.



Komponenter

Parallellt med objektorienteringen kom komponentbaserad utveckling som byggde på att skapa ett gränssnitt/kontrakt till klienten som komponenten förhöll sig till utan att dela med sig av sin implementation (Interfaced Based Programming). Därmed byttes implementationsarv ut mot aggregat av komponenter. Det gav lösare koppling mellan klienten och komponenten och man kunde skapa flexiblare system.

SOAP

Fortfarande fanns det dock problem med komponentbaserad utveckling. Bland annat så fanns det flera olika standarder för protokollen som användes för distribuerad kommunikation. Gränssnitten mot komponenterna kunde inte delas mellan olika plattformar på grund av datatypers olika representation. Så istället för att enas om en gemensam plattform, ett programspråk eller en komponentmodell så enades man om ett gemensamt protokoll även känt som SOAP.

Web Services

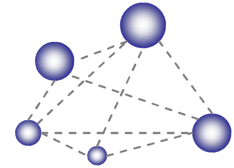
Efter SOAP 1.0 har protokollet utvecklats och det betraktas nu som en W3C-standard. Aspekter på protokollets betydelse för återanvändning och flexibilitet har fått stor betydelse. Bland annat finns standarder för hur SOAP ska användas för säker kommunikation (WS-Security), i en kontext (WS-Transaction), för att garantera leverans (WS-ReliableMessaging). Allt detta känner vi idag som Web Services.

Web Services är det nya sättet att koppla samman moduler och praktisera återanvändning och flexibilitet. En Web Service är lösare kopplat än komponenter eftersom typsystemet (XML Schema) är standardiserat och flexibiliteten garanteras med hjälp av kontrakt definierade i WSDL (Web Service Description Language).

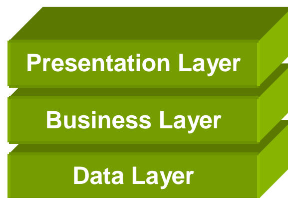
Visioner inom systemarkitektur

Vi har lärt oss att bygga applikationer i lager för att göra designen flexibel och utbytbar. Klient/server blev ett begrepp och bygger på fleranvändarstöd som saknades i vid den tiden. En serverprocess kunde nu dela ut sina resurser till flera klienter och därmed öka skalbarheten. Skalbarheten skedde då på bekostnad av utbytbarheten eftersom mycket av applikationslogiken byggdes in i klienten.

3-skiktade lösningar aktuella samtidigt som komponentbaserade teknologier blev tongivande. Utbytbarheten förbättrades eftersom logiken separerades från klienten och fick sitt eget lager. Efter det har vi levt med flerskiktade arkitekturer för att just främja utbytbarheten genom att abstrahera lager på lager. Det ger oss möjlighet att exempelvis

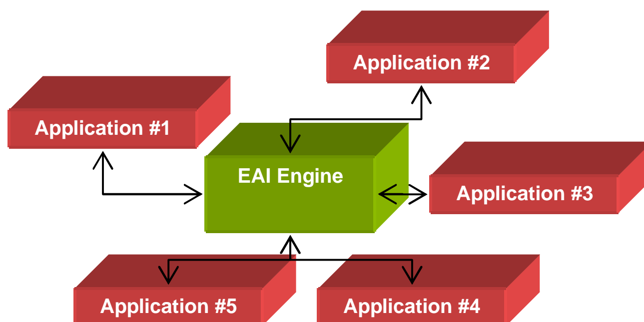


byta databas då ovanliggande lager abstraherar bort beroendet till en specifik produkt. Tekniska lager i arkitekturen fokuserar på utbytbarhet i form av leverantörsspecifika produkter och plattformar, inte av applikationer, moduler eller system.



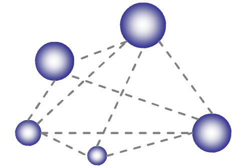
Figur 4 Skiktad arkitektur för flexibel arkitektur.

Utbytbarhet och flexibilitet har varit tongivande inom arkitektur och design, medan återanvändning inte fått så stort fokus förrän på senare år och då med systemintegrationens intåg. Integration handlar om arkitekturval där system och applikationer återanvänds och samspelar med varandra för att dela information och tillsammans vara delaktiga i processflöden. Det typen av integration är ortogonal mot flerskiktade arkitekturer och fokuserar mer på flexibilitet i ett vertikalt synsätt. Att dela information mellan applikationer och system i en IT-miljö kallas för Enterprise Information Integration (EII) och samspelet i en process kallas Enterprise Application Integration (EAI).



Figur 5 EAI kopplar samman monolitiska applikationer.

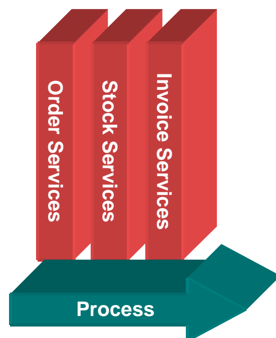
EAI utgör en stor del i utvecklingen av SOA, vilket vi snart kommer att se. Problemet med EAI är att det bygger på samspel mellan applikationer och system, vilka inte är tillräckligt små enheter för att få ut en bra utbytbarhet. Det andra problemet är att varje system och applikation har sina egna sätt att integreras. Man kan se EAI som en övergångsperiod tills dess SOA slagit igenom. Problemet med EAI är att det även kan vara en SOA-killer eftersom det kostar så pass mycket att införa idag utan att på sikt ge några förbättringar för verksamheterna i jämförelse med SOA.



Visioner inom affärs- och verksamhetsutveckling

Verksamhetsutveckling praktiserar återanvändning och flexibilitet som ett redskap för att effektivisera processer och för att kunna dra ned kostnader i ett företaget genom en omorganisation eller för att omstrukturera delar av verksamhetsprocesserna. De vill ha flexibilitet i affärsverksamheten och ser helst att det avspeglar sig i systemen de använder.

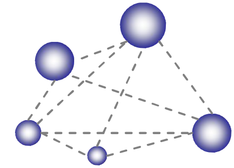
Företag har definierade flöden (processer) som kopplar samman ett företags tjänster i verksamheten. Det kan exempelvis vara order-, lager- och faktureringsprocessen eller anställningsprocessen. Dessa processer använder sig av tjänster som olika delar av organisationen eller externa parter ställer till förfogande.



Figur 6 Tjänster är ett vertikalt synsätt som knyts samman av en process.

Den stora utmaningen för ett företag är att göra förändringar i processerna utan att de medför för stora omkostnader. Säg att ett företag vill flytta sin fakturering till ett externt företag som gör det billigare än att sköta faktureringen internt. Den effektiviseringen är möjlig om de kan bryta ut faktureringsprocessen från order, lager och faktureringsprocessen. Beroende på hur organisationen och IT-stödet ser ut så kan det vara mer eller mindre smärtfritt. En organisationsförändring är vardagsmat i en verksamhet och genomförs ständigt i växande och föränderliga företag. Sitter däremot företaget på monolitiska IT-system där alla delar hänger samman och inte kan ersättas så kommer kostnaden inte att betala sig, även om det skulle vara gratis att fakturera genom den externa tjänsten.

En verksamhet där man erbjuder tjänster till kunder och partners samtidigt som de nyttjar tjänster hos leverantörer skapar en lång värdekedja. Företag erbjuder tjänster, men nyttjar även tjänster i sin verksamhet för att fungera. I vissa fall kan tjänsterna de nyttjar vara en del av företaget, men de kan även använda sig av externa leverantörer.



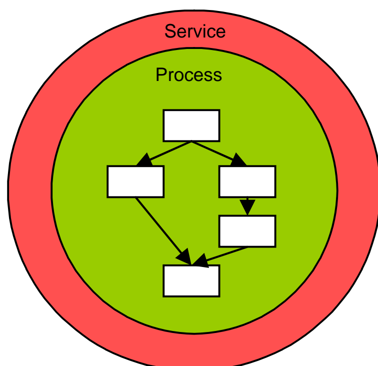
Visionerna delar samma mål

Oavsett om det gäller systemutveckling, arkitektur eller verksamhetsutveckling så delas samma mål. Alla tre strävar efter lägre kostnader, flexibilitet, utbytbarhet och återanvändning. Om man väver samman de tre gruppernas målsättning så borde det bli något i stil med; En arkitektur som bygger på samverkan mellan små självfungerande tjänster som är definierade av verksamhetsprocesser och baserade på standardiserad teknologi. Vi kallar det SOA från och med nu!

SOA är alltså en arkitektur som tar hänsyn till alla inblandades krav på flexibilitet och kostnad.

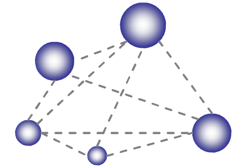
Tjänster och processer

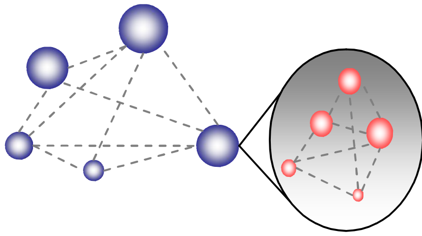
I en tjänsteorienterad värld är förhållandet mellan en tjänst och en process samma sak som förhållandet mellan ett interface och dess implementation i en komponentbaserad värld. Tjänsten är kontraktet som beskriver under vilka förutsättningar processen exekverar. Det kan exempelvis gälla hur informationen skall se ut som tjänsten behöver och i vilken form den ska levereras.



Figur 7 En tjänst beskriver under vilka förutsättningar en process kan exekvera.

I processen beskrivs samverkan med andra tjänster, som i sin tur kan vara delprocesser. En process kan vara en del av en annan process eller bara sin egen, helt ovetande om högre ordnade processer. Det ställer kravet på att tjänster fungerar som applikationer, d v s att de inte är beroende av eller får vara beroende av en anropande process (tillskillnad från komponenter som nästan förutsätter att vara en del av en applikation). Det medför att här slutar likheten mellan SOA och komponenter och därmed kan vi ta död på myten om att SOA har med komponenter som görs tillgängliga på Internet.





Figur 8 En process av tjänster som i sin tur kan innehålla andra processer.

Alla tjänster inom en process kan dock inte betraktas som processer. Ta exempelvis faktureringsprocessen som kanske behöver slå upp adressinformation innan fakturan ska skrivas ut. Då behöver den använda sig av en tjänst som fungerar som ett registeruppslag. Det är en nod i processen där inte processen kommer att nyttja en annan delprocess. En tjänst som är en nod skiljer sig från de tjänster som implementerar en egen process. De är oftast inte asynkrona utan alltid närvarande och kan anropas när som helst.

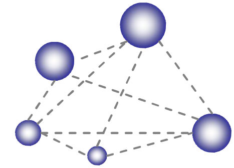
En tjänst som implementerar en process däremot måste ibland betraktas som frånvarande. Då kan man inte sitta och vänta på svar som kan komma mycket senare och ibland inte alls som man tänkt sig. I faktureringsprocessen kommer fakturabetalningen mycket senare och där måste processen kunna vänta ut svaret eller fortsätta och vara beredd på att det kommer in senare för att då kunna hantera det som en del i processen.

Web Services och ESB

Tjänster använder kontrakt som beskriver dess användning och funktion. Eftersom samma analogi finns i Web Services så är det oftast ett mycket bra sätt för att implementera tjänster. Ett WS-kontrakt beskrivs med hjälp av WSDL som innehåller en logisk del som beskriver tjänstens funktion och en fysisk del som beskriver användningen av den. Tjänster som använder andra tjänster kallas i sin helhet för en bussarkitektur. En sådan buss bygger på att tjänster använder och används av andra tjänster i ett arbetsflöde. Arkitekturen kallas för Enterprise Service Bus (ESB).

SOA förutsätter funktionaliteten som en service bus erbjuder. En ESB karaktäriseras bland annat ha support för:

- WS-*
- WSDL, SOAP och UDDI



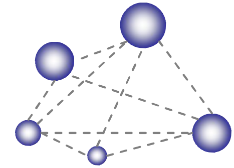
- Workflow (BPEL) och support för LRT (Long Running Transactions) genom dehydration och rehydration.
- Asynkron meddelandehantering
- Fel- och valideringshantering
- Adapters för att länka in existerade applikationer
- Lastbalansering
- Failover
- Transformationer mellan meddelandeformat
- Aktivitetsmonitorering (Business Activity Monitoring)

Standardiserad säkerhetsmodell för att autentisera, auktorisera och granska (audit)

En ESB-motor är en perfekt produkt för att implementera processorienterade tjänster. Varje processbeskrivning man skapar i en ESB kan exponeras som en tjänst och dra nytta av andra tjänster både i och utanför ESB. Vissa ESB-motorer bygger enbart på Web Services (True SOA) och kommunicerar därför enbart med tjänster som beskrivs med hjälp av WSDL. Själva processen eller arbetsflödet (Workflow) implementeras genom stödet i ett språk som heter Business Process Execution Language (BPEL).

BPEL beskriver för ESB-motorn hur processen arbetar och hänger samman med andra tjänster. Processen byggs upp logiska konstruktioner och sekventiella och parallella flöden. Med If-satser där processen gör ett val, While-satser där processen hanterar en loop eller händelser där processen väntar på att något ska hända. Alla dessa konstruktioner i BPEL kan oftast modelleras genom ett grafiskt verktyg som sedan mappar till BPEL. Det gör processen mera överskådlig och är enklare att ändra.

En ESB-motors största uppgift förutom att exekvera BPEL är att kunna hantera tillstånd. Tillstånd i en process ska inte blandas ihop med tillstånd i en tjänst. Tjänster är tillståndslösa och ska inte ändras från ett anrop till ett annat, såvida det inte är under en transaktion. Om man tar faktureringsprocessen så kommer den att behöva bevara sitt tillstånd undertiden som den väntar på svar från betalningstjänsten som den använder. Det kan ta månader innan en kund betalat sin faktura och under tiden kommer processens instans att behöva bevaras.



Slutsats

Hoppas den här artikeln har visat var behovet av SOA kom ifrån och var vi är idag. Nästa steg är att börja tänka processorienterat och använda ESB som en naturlig del i IT-miljön.

SOA kommer att förändra mycket inom mjukvaruutvecklingen och för att kunna komma framåt måste vi börja arbeta mer praktiskt med SOA. Hittills har man mest blivit matad med metaforer och teorier, men nu finns verktygen så det är bara att sätta igång.

EAI är enligt min mening bara ett sjukdomstillstånd och inte en arkitektur. EAI är dyrt och på sikt inte heller en god investering. EAI försöker spela samma spel som SOA men med alldeles fel kort på handen.

Nedan finns länkar till implementationer av ESB som man kan ladda ned och börja använda:

Cape Clear ESB

ActiveBPEL (open source)

Process eXecution Engine

Notera att flera större leverantörer som Microsoft, IBM, BEA och Oracle också snart kommer att integrera ESB-stöd i applikationsservrar och bryta ut det från tunga EAI-produkter.

Jonas Ekström, jonas@swenug.com