Container-konceptet

Att bygga en arkitektur för förändring

IT-arkitekt 10, projektarbete

Martin Bergström

Version 1.00, 2002-01-08

Container-konceptet

Att bygga en arkitektur för förändring

Sammanfattning

Många större företag som har använt IT under lång tid, har skaffat sig en IT-miljö som är komplex och heterogen. De flesta systemen i miljön har ofta byggs som om de vore de enda där, utan hänsyn till övriga system. Storleken och komplexiteten gör nu att miljön blir svår och dyr att underhålla och förändra.

På marknaden idag finns två stora grundidéer, MS .NET [.NET] och Javas J2EE [J2EE]. Båda täcker väl de flesta behov som kan tänkas finnas för nyutveckling. Men, de tar lite hänsyn, dels till varandra, dels till de system som redan finns, dels till ett framtida tänk eftersom det troligen inom nåt år kommer någon ytterligare stor grundidé om hur system borde byggas.

Det saknas idag ett tänk för systemen under hela dess livstid. De flesta tänk gäller bara hur man bygger systemen och får delar att kommunicera med varandra men saknar information om hur man till exempel byter ut driftsatta delar, bygger bra förvaltningsbara system, system som kan förändras utan stora synkroniseringsproblem etc.

Detta dokument beskriver två grundidéer, Container-konceptet och ett tjänstekoncept, som båda syftar till att skapa en övergripande arkitektur som får befintliga och nya system att samverka i en heterogen miljö. De gör arkitekturen mer förändringsförberedd och tar bort eller förminskar många av de förändringsproblem som många system har idag. Koncepten kan användas till både enbart enskilda system som till en hel stadsplanering. I dokumentet ges exempel på båda.

Container-konceptet är en ide om hur komplexa data kan hanteras på ett smidigt sätt. En Container är en paketering av ett antal variabler som att de hanteras som en enhet. En Container kan innehålla alla typer av variabler, inklusive andra Containers för att därmed skapa en trädstruktur.

Tjänstekonceptet bygger på en idé om att applikationer skall fungera som tjänster, det kan vara hela applikationer eller delar av den. Det kan också användas som en skiktningsmetod. Tjänsten erbjuder en återanvändning av funktionalitet, också ett sätt att få skikten att bli mer oberoende av varandra.

Koncepten är plattforms och teknikoberoende och bygger på holistisk grundsyn. En av de stora fördelarna med dem är att de kan smygas in i en befintlig miljö utan stora omskrivningar eller investeringar.

**Syftet med dessa två koncept tillsammans är
att om systemen är uppbyggda helt eller delvis efter dessa tankar,
så ger det en arkitektur som är förberedd för framtida förändringar.**

Innehållsförteckning

SAMMANFATTNING **2**

INLEDNING **4**

BAKGRUND 4
PROBLEM 5
AVGRÄNSNING 5
MÅLGRUPP 5
METODVAL 6

GENOMFÖRANDE **7**

CONTAINER-KONCEPTET 8
Är Containers XML ? *8*
Ursprung *8*
Fördelar med Containers *9*
TJÄNSTE-KONCEPTET 11
Web Services *12*
Övergripande implementation *13*
MÅLBILD 16
SCENARIOR 17
Förändrad fältlängd *17*
Ny funktionalitet *18*
Ny visualisering *19*
Kommunikation mot befintligt system *20*
KVALITETSATTRIBUT 21

SLUTSATSER /DISKUSSION **24**

Fördelar *24*
Nackdelar *24*
Framtid *25*

REFERENSER **25**

APPENDIX **26**

A: EXEMPEL SCHENKER CTTS 26
B: EXEMPEL BIBLIOTEKSSYSTEMET LIBRA 30
C: NAMNFORMER 32
D: SERIALISERINGSFORMAT 33
E: FELHANTERING 36
Schenkers felkoder vid svar från en tjänst *36*

Inledning

Bakgrund

Många större företag har idag en heterogen IT-infrastruktur som liknar figur 1. Den innehåller många olika plattformar från flera leverantörer, i näten går en mängd olika protokoll, etc. Applikationerna som körs i infrastrukturen är levererade av ett flertal leverantörer och de är skrivna i olika språk, baserade på olika paradigmer och från olika tidsåldrar.

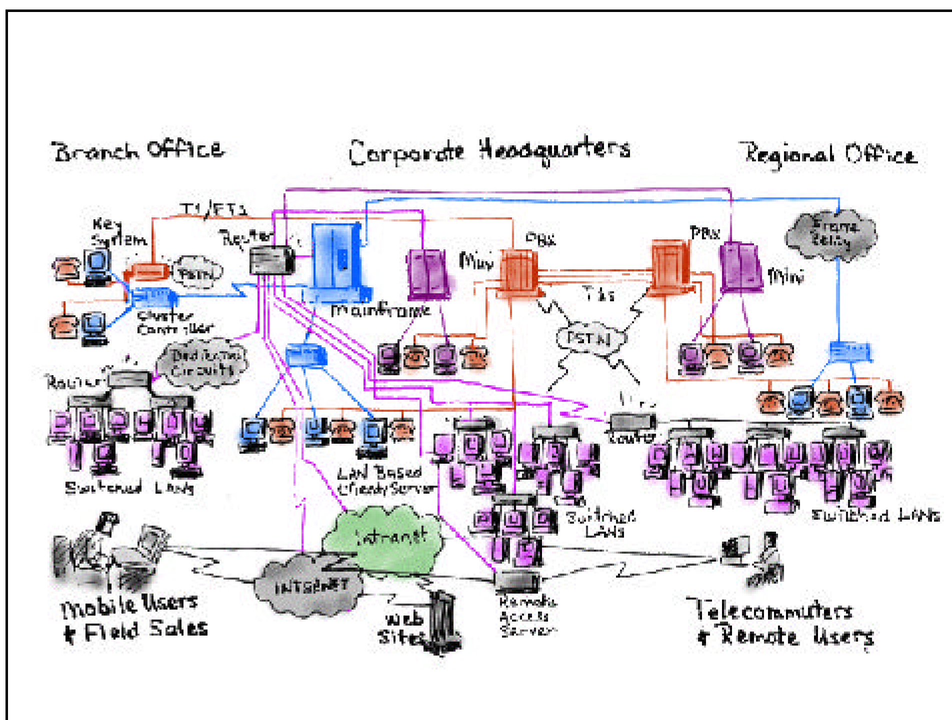
Efter Webbens intåg i infrastrukturen ställs det förändrade och större krav på den än tidigare. Nu skall applikationerna och infrastrukturen:

- klara större öppethållande än tidigare
- systemen skall kunna integreras mer med varandra
- systemen skall kunna erbjuda nya klienter (browser, wap-telefon, etc)
- hantera okända, oerfarna och ett otal användare

samt att förutsättningarna förändras allt snabbare.

Komplexiteten i den befintliga miljön gör den svårarbetad, smärre ändringar kan få stora konsekvenser och det blir mycket dyrbart med långsamma leveranser som följd.

För många företag idag är det heterogena system som gäller. Och detta gäller troligen även i framtiden. Målet är att hitta en grundarkitektur som är plattforms-, programspråksoberoende och produktoberoende som är förberedd för och underlättar integrering och förändringar.



Figur 1 Exempel på heterogen miljö

Problem

Är det möjligt att bygga en heterogen arkitektur så att den är förberedd på förändringar ? Är det möjligt att skapa den nya förändringsbarheten i den befintliga ?

Många system är idag byggda som flerskiktad client/server. Detta byggsätt har många fördelar men en av de större nackdelarna är det synkroniseringsproblem som uppstår när förändringar sker i servern, vilket ofta kräver förändringar i klienterna, och servern och klienterna måste uppdateras samtidigt. Speciellt svårt blir det när klienterna körs utanför den egna domänen, ex hos kunder. Är det möjligt att bygga client/server-system på ett sådant sätt att detta problem förenklas eller upphör ?

Återanvändning är idag ett hett ämne. En av de större vinsterna med återanvändning vore om det gick att återanvända den befintliga arkitekturen och de system som körs där. Är det möjligt att återanvända befintliga system så att nya system kan skapas ?

I distribuerade system, ex uppbyggda som tjänster, ger problem med spårbarheten, ex när ett fel uppstår och en systemadministratör skall tolka vad som gick fel och varför. Är det möjligt att bygga systemen så att spårbarheten underlättas och att loggarna ger så mycket information att felet kan återskapas ?

Många tillägg som görs, baseras på någon händelse i ett befintligt system, ex att kunna hämta ut fakturainformation på den information som faktureringen sänder till reskontran för att kunna erbjuda en bättre statistik än vad de befintliga systemen kan. Är det möjligt att enkelt kunna koppla mitt nya system till denna typ av händelser ?

För att kunna lösa dessa problemområden behövs ett övergripande synsätt på hur information skall hanteras. I detta dokument beskrivs Container-konceptet som är en grundsyn på hur man kan hantera komplexa data med på ett plattforms- och programspråksoberoende sätt. I dokumentet beskrivs också ett tjänstekoncept baserat på Container-konceptet. Dessa koncept har ett antal synergieffekter som ger ett flertal lösningsförslag på ovan nämnda problemområden.

Avgränsning

I detta dokument kommer Container-konceptet att beskrivas på ett övergripande plan för att ge en allmän förståelse för hur komplexa data kan hanteras. Dokumentet kommer att beskriva hur ett tänkt tjänstekoncept kan vara uppbyggt baserat på Container-konceptet. Exempel på implementationer ges i appendix.

Målgrupp

Målgruppen för detta dokument är främst de som är ansvariga för en heterogen infrastruktur, såsom:

– IT-chefer
– IT-arkitekter – stadsplanerare

Men också:

– IT-arkitekter – systemdesigner
– Systemutvecklare

Metodval

Första gången Container-konceptet användes var slutet 80-tal, för att kunna hantera de komplexa data som skall finns i ett bibliotekssystem. Konceptet har använts i praktiken i över tio år med mycket goda resultat i ett flertal system. Detta dokument är en sammanfattning av många års personliga erfarenheter för att kunna lösa en hel del av de problem som idag finns i stora heterogena arkitekturer.

Lösningarna finns idag implementerade i befintliga system men inte alla lösningarna i ett och samma system. Därför finns det goda erfarenheter av alla förslagen var för sig.

Detta dokument kommer att ligga till grund för en framtida stadsplan.

Genomförande

Många större företag som har använt IT under lång tid, har skaffat sig en IT-miljö som är komplex och heterogen. De flesta systemen i miljön har ofta byggs som om de vore de enda där, utan hänsyn till övriga system. Storleken och komplexiteten gör nu att miljön blir svår att underhålla och förändra.

På marknaden idag finns två stora grundidéer, MS .NET [.NET] och Javas J2EE [J2EE]. Båda täcker väl de flesta behov som kan tänkas finnas för nyutveckling. Men, de tar lite hänsyn, dels till varandra, dels till de system som redan finns, dels till ett framtida tänk eftersom det troligen inom nåt år kommer någon ytterligare stor grundidé om hur system borde byggas.

Problemet med att det inte finns något övergripande tänk för heterogena miljöer, är att det skapas en mängd olika systemöar i IT-miljön. I och med webbens intåg, ställs idag större krav än tidigare på att systemen skall kunna integreras med varandra, vara mer öppna, kunna visualiseras på en mängd nya sätt etc.

Det saknas idag också ett tänk för systemen under hela dess livstid. De flesta tänk gäller bara hur man bygger systemen och får delar att kommunicera med varandra men saknar information om hur man till exempel byter ut driftsatta delar. Om man synar sina system i backspegeln, kommer man fram till att de:

– Har lång livslängd (längre än man trott från början)
– Har kommit i många versioner
– Blivit integrerade – förr eller senare - med andra system
– Kommer att bli knepiga att ersätta
– Har blivit svåra och dyra att underhålla
– Har fått sina förutsättningarna radikalt förändrade

För att till viss del råda bot på detta, presenteras i detta dokument två grundidéer, Container-konceptet och ett Tjänste-koncept. Dessa idéer kan användas allt från en övergripande nivå, ex för att hantera kommunikation mellan system, till att hanteras genomgående inom ett system. Koncepten kan implementeras på en mängd olika sätt, dels för att olika miljöer påverkar implementationen, dels olika kvalitetskrav såsom prestanda, modifierbarhet kan ha sin påverkan. Om implementationerna bygger på samma grundide, så kommer de också att uppföra sig på samma sätt.

Dokumentet beskriver först övergripande de två koncepten, därefter den målbild som de skall stödja plus några vanliga scenarior. Därefter en reflektion kring ett antal kvalitetsattribut. I appendix finner Du exempel på två implementationer.

Container-konceptet

Container-konceptet är ett tänk kring hur data, främst komplexa, kan hanteras. Min definition av komplexa data är när det bäst beskrivs i en trädstruktur istället det traditionella poster och fält vilket leder till att alla system hanterar komplexa data fast i olika omfattning.

Containern är en paketering av en eller flera variabler (fält), på ett sådant sätt att alla variablerna kan hanteras som en enhet. Istället för att sända 5 variabler till en metod, sänds istället en Container med de 5 variablerna i. En Container kan – i teorin – hantera obegränsat med variabler, vilket gör anropet mycket enklare.

En Container kan hantera alla typer av variabler, allt från heltal och strängar, till vektorer, dokument, binärer (bilder, program) till containrar.

En Container har ett antal egenskaper:

- **Innehåll** - Krav finns endast på obligatoriska data. Dessa skall vara så få som möjligt. En Container kan vara tom om inga obligatoriska fält är definierade. Fält som saknar värde behöver inte men kan existera. Den får innehålla mer information än vad mottagaren vill ha. Ej önskad information skall ignoreras (dvs det är inget fel). Kan innehålla många poster / objekt (I princip obegränsat med poster och fält)
- **Default data** – Avsaknaden av värde i ett fält kan också vara värdebärande om man har definierat ett default-värde för det fältet.
- **Struktur** – Den har struktur men inget krav på ordning, ex fälten customer_number och customer_name kan komma i vilken ordning som helst.
- **Självbeskrivande** - Baserad på en associativ vektor (exempel är hashtable, dictionary, dom, xml). Den innehåller fält=värde, ex: customer_number=9999, customer_name=Ementor AB. Värdet kan vara vilken datatyp som helst, ex strängar, tal, containrar, dokument, binärer, såsom bilder eller program.

Containers innehåller metadata som anger:

- Vilka fält som ingår (ordningen oväsentlig)
- Hur långt ett fält och dess typ är (exakt format oväsentligt)
- Hur många gånger det förekommer (multipla förekomster tillåtet, även noll)
- Fältinnehållet kan vara en annan container (möjliggör trädstrukturer)
- Fältinnehåll kan vara dokument, ritningar, mm

ÄR CONTAINERS XML ?

Nej det är det inte. Container-konceptet är ett teknik och implementations-oberoende koncept. Däremot så är XML ett utmärkt sätt att implementera en Container på, exempel på sådan implementation finns i appendix. Som regel kommer Containern troligtvis att ha flera olika implementationer beroende på var den används. Ett exempel baserat på HQF (HTTP QueryString Format) visas också.

Alla plattformar kan ännu inte använda XML fullt ut. Det finns också tillfällen där det inte är praktiskt att använda XML, ex pga prestanda. Det är viktigt att Containern bygger på samma grundidé oavsett implementation, så att det är enkelt att konvertera mellan olika format. Det skall alltid vara ett 1-1 förhållande mellan implementationerna.

URSPRUNG

Ursprungsidén till Container-konceptet är från slutet 80-tal när jag byggde ett bibliotekssystem [Libra]. Systemet blev en succé och körs idag av folkbiblioteken i ca hälften av Sveriges kommuner + företag och institutioner.

Ett bibliotekssystem har en mycket komplex datastruktur för att beskriva ett media. För att kunna hantera ett media, ställs vissa grundläggande krav på databasen:

– Varje fält måste kunna ha obegränsad längd (vill någon mata in hela romanen i ett fält, måste systemet klara detta)
– Varje fält måste kunna upprepas i oändligt antal gånger (finns 17 författare skall alla med)
– Databasen måste kunna hantera obegränsat med fält per media (teoretiskt behövs drygt 1000 fält för att kunna beskriva olika media, ett media kan vara en bok med tillhörande CD och nothäften)
– Fält måste kunna grupperas för att bilda en enhet (ex författarens namn, födelseår, födelseort, mm för att kunna skilja en författare från en annan med samma namn). Söksystemet måste kunna söka/sortera på enskilt fält eller en enhet.
– Biblioteket måste kunna få definiera sina egna fält och fältgrupperingar, tillägg tillåts under drift (runtime)
– Ett fält skall kunna ha flera sök/sorteringsfält knutna till sig (ex författarens namn måste registreras som det står i mediet, medan författaren kanske stavas vanligtvis annorlunda i Sverige. I en mycket stor databas hittades 40 olika stavningar på Moamar al-Khadaffi)
– Det skall klara hela latinska alfabetet, kyrilliska, grekiska plus specialtecken såsom matematiska etc

De verktyg som då stod till buds klarade inte av att på ett smidigt sätt hantera denna komplexitet och därför uppstod Containern, ett sätt att hantera komplexa data. Det uppstod ett antal synergieffekter av att vi började använda containern, bla enklare underhåll, mycket kompakt lagringsformat (tomma fält existerar inte), enkelt att serialisera för att kunna spara datat eller distribuera det till ett annat system, enklare visualisering, förenklar spårbarhet etc.

FÖRDELAR MED CONTAINERS

Genom sin konstruktion så blir Containern ett mycket starkt och flexibelt redskap. Styrkan ligger främst inom följande områden:

Fråga / svar

Detta är det i särklass vanligaste anropssättet idag, ex allt ifrån att ett system anropar ett annat till att någon del i programkoden anropar en metod eller funktion. Ett stort problem med detta sätt är att om svarsdelen förändras så måste i regel även anropsdelen också förändras.

Till exempel, säg att en funktion anropas med 3 variabler och returnerar ett heltal.

```
int question(char *var1, int var2, double var3)
```

Den som anropar denna funktion, måste anropa med rätt antal variabler, rätt typning och rätt ordning för varje variabel. Ändras ordningen, typningen eller antalet så måste den som anropar också ändras. Och ändringen måste ske samtidigt. Inom Client/server-världen skapar detta ett enormt synkroniseringsproblem i samband med uppdatering av programvaran, speciellt om de anropande systemen är utanför den egna kontrollen, ex hos kunder.

Med Container-konceptet kan denna anropsform ändras, så att anropet sker istället med en Container och returnerar en Container. Med denna lösning upphör helt detta uppdateringsproblem eftersom antalet argument alltid är detsamma, det ändras aldrig.

```
HashTable * question(HashTable * Container)
```

Ytterligare ett problem uppstår när antalet argument överstiger ca 10, då blir traditionellt anropssätt ohanterlig. Även om en Container innehåller tusentals fält, så blir anropet likväl detsamma.

Lösa skikt

Många pratar idag om att det skall vara lösa skikt men få beskriver hur dessa skall se ut eller fungera. Ett löst skikt för mig är ett oberoende skikt, dvs att den som anropar och den som svarar skall kunna göra det oberoende av varandra. Container-konceptet erbjuder enligt mitt tycke den starkaste lösningen för hur lösa skikt skall vara uppbyggda. Den erbjuder stora förändringsmöjligheter på både anropande som svarande sida utan att de behöver påverka varandra

Modifierbarhet

Genom sin flexibilitet är det mycket enkelt att lägga till och ta bort fält eller ändra dess längd eller typning. Eftersom det är tillåtet att Containern innehåller information som inte är definierad av båda parter, så kan båda parter lägga till information efter eget tycke, såsom mer information pga ökad funktionalitet eller debuginformation etc.

Storlek

En Container kan innehåller andra Containers. Det gör att en Container i princip skulle kunna innehålla en hel databas. Ett exempel på detta är en utsökning. Funktionen som lämnar svaret kan lämna hela informationsmängden i retur, istället för som normalt att en post skall hämtas åt gången. Det leder till att oavsett svarsmängdens storlek på går det åt bara en fråga och ett svar för att hämta den.

Generella rutiner

Eftersom Containern paketerar alla variablerna, så kan de alla hanteras som en enhet med generella funktioner (beskrivs mer i detalj senare), ex

- Serialisering till olika format / mellan format
- Kommunikation, Kryptering, Komprimering
- Konvertering
- Loggning
- Felhantering och Validering
- Sparas i databas eller i fil
- Visualisering
- Prenumeration

Avobjektifiering av affärsobjekten

En av objektorienteringens styrkor är också en av dess nackdelar, detta att funktionalitet och data är knutet ihop i ett objekt, vilket ställer till problem när man skall kommunicera med en icke objektorienterad värld.

Genom att allt data passerar genom ett containerobjekt, döljs detaljer om affärsobjekten för omvärlden. Mottagaren kan ha sin egen (förenklade) modell av dessa och göras relativt oberoende av den centrala affärsfunktionen. Mottagaren behöver inte vara ett objektorienterat system över huvud taget, utan kan vara ett gammalt system utan några affärsobjekt.

Detta är en förutsättning för att kunna migrera dagens legacyvärld med en objekt och komponentorienterad värld

Tjänste-konceptet

Tjänste-konceptet bygger på idén att funktioner/metoder, skikt, applikationer etc (oavsett nivå) skall erbjuda tjänster istället för bara funktionalitet. Skillnaden är mest i synsätt och inte så stor vad gäller implementation. Dock ger denna skillnad en hel del fördelar gentemot traditionellt sätt att bygga system.

Skillnaden förenklar bla integrerbarhet, modifierbarhet, återanvändning men leder också till att styra upp hur saker är uppbyggda, vilket också leder till återanvändning av kod, mallar, felhantering, loggning, spårbarhet etc.

För att en funktion skall kallas en tjänst, ställs ett antal krav:

- **Löst skikt** – Tjänsteuppbyggnaden är en skiktningsmekanism och den skall vara uppbyggd med ett löst skikt, den som anropar och tjänsten skall kunna agera oberoende av varandra.
- **Kommunikation** – Det skall gå att anropa tjänsten via ett antal olika kommunikationssätt (eller i alla fall implementera flera med en relativt rimlig arbetsinsats)
- **Spårbarhet** – Eftersom den anropande och tjänsten agerar oberoende av varandra, ställer det högre krav på felhantering och den spårbarhet som behövs för att kunna spåra felet och dess uppkomst. Helst skall indata sparas så att det går att återskapa felet.
- **Loggning** – Det skall gå att studera anrop och svar från en tjänst
- **Validering** – Den måste validera sitt indata, den får inte förlita sig på att den anropande gör rätt.

Det är önskvärt att tjänsten också erbjuder

- **Testbarhet** – Det bör gå att dels testa själva tjänsten, dels erbjuda test av kommunikation mot tjänsten utan att det påverkar produktionsdata.
- **Prenumeration** – Det bör gå att prenumerera på ett anrop eller ett svar. Det ger stora möjligheter att återanvända information och händelser som finns i de befintliga systemen.
- **Status** – Tjänsten bör kunna hålla reda på trafikinformation om sig själv, ex antal anrop OK, antal anrop fel, snitt- och max-tid för bearbetning, hur länge den varit igång, hur mycket resurser den tagit i anspråk etc.

En tjänst uppbyggd på Container-konceptet kan mycket enkelt implementera dessa krav. Det sker genom att det byggs en fasad med denna funktionalitet inbyggd mot en eller flera bakomliggande funktioner som skall fungera som tjänster. Eftersom fasaden erbjuder all denna funktionalitet, så behöver inte det bakomliggande kunna hantera Containers. Det ger möjligheten att befintliga system eller ex Cobol-miljöer kan fungera som tjänster.

Alla tjänster baseras på samma arkitektur- och design-mönster, mer beskrivna av [Buschmann]

Arkitekturmönstret består av:

- **N-tier client/server** med lösa skikt (baserat på containrar) - Varje skikt har samma gränsyta och anropssätt, även det skikt närmast klienten har samma gränsyta. Tanken är att alla delar anropas på samma sätt och kan därmed enklare återanvändas. Eftersom alla delar har samma gränsyta, är ingen sist utan det gör att det går att bygga på med fler skikt utan förändring, dvs inget skikt får anses vara det sista. Anropen är av typen Fråga/svar. Denna lösning ger en större skiktning än vad som normalt beskrivs. Varje del i systemet kan vara uppbyggd som en tjänst till övriga systemet (en form av komponentifiering).
- **Broker** – Erbjuder möjligheten att hitta tjänsten utan att kunna dess exakta adress

Designmönstret för en tjänst består av en kombination av flera mönster:

- **Fasad** – alla tjänster har en fasad som gör att den som anropar inte kan se skillnad på en tjänst på en stordator eller på en NT, oavsett programmeringsspråk etc. Fasaden är uppbyggd i ett flertal delar, med: kommunikation, konvertering, validering och en broker och övervakas av en Manager som hanterar fel, spårbarhet mm.
- **MVC** – Hanterar visualiseringen skild från datahanteringen
- **Prenumeration** – Managern använder en prenumerationstjänst för hantera fel, spårbarhet men erbjuder som synergieffekt möjlighet för utomstående att ta del av informationsflödet.

WEB SERVICES

Hur förhåller sig detta tjänstekoncept gentemot det tjänstekoncept som alla pratar om idag, Web Services. Vid en första anblick egentligen ingenting, vid en djupdykning en hel del.
Först en liten resumé om vad Web Services är. Det bygger på UDDI (Universal Description, Discovery and Integration) [UDDI] som i sin tur bygger på tre delar:

- **Kommunikation** – Bygger på SOAP [W3C] (Simple Objekt Access Protocol), ett kommunikationssätt och format, använder XML som format och främst HTTP som protokoll men kan använda andra.
- **Gränssnittsbeskrivning** – Baseras på WSDL [W3C] (Web Services Description Language), ett sätt att beskriva XML-meddelanden
- **Katalogtjänst** – Ett ställe där gränssnittsbeskrivningarna finns lagrade och tjänsten erbjuder utsökning enligt ett flertal olika kriterier, ungefär som vita och gula sidorna i en telefonkatalog.

Grundidén är att katalogtjänsten erbjuder adressen till en tjänst samt en beskrivning på hur meddelandet ser ut. Kommunikationen mot tjänsten sker sedan med SOAP.

Det går mycket väl att använda Web Services som implementation för tjänstekonceptet. Om jag nu väljer att använda Web Services, får jag då lösa skikt, slippa synkroniseringsproblemen, få hög förändringsbarhet etc ?

Nej, det får Du inte utan vidare. Trots att Web Services använder XML, trots att det bygger på ett tjänstekoncept, så är det i sin grund fortfarande för stark kontroll, ordning och reda och har egentligen inte löst någonting mer än att det erbjuder ytterligare ett sätt att kommunicera i distribuerade system. SOAP kommer att ersätta DCOM i Microsofts nya .NET [.NET].

Grunden i SOAP är att kunna göra ett metodanrop med flera variabler mot en distribuerad metod, det ger samma bekymmer med uppdatering på båda sidor pga ändring av antalet variabler, dess ordning och dess typ. Hur skall jag göra om jag vill använda Web Services som bas för Tjänstekonceptet baserat på ett Container-koncept ? Gör endast följande:

- **Distribuera en Container** – Använd SOAP som protokoll men sänd endast en variabel fram och tillbaka, dvs en Container.
- **Skapa en övergripande Tjänst** – Skapa en fasad enligt ovan beskrivning och låt SOAP anropa den istället för metoden direkt.
- **Ha egen felhantering** – Använd egen felhantering enligt detta dokument och använd SOAP:s enbart för att hantera själva kommunikationen

ÖVERGRIPANDE IMPLEMENTATION

```
Element exampleService(Element aContainer)
       {
       Error     error      = new Error();
       Subscribe subscribe  = new Subscribe(exampleService);
       Validate  validate   = new Validate(Element validateSchema);
       Convert   convertIn  = new Convert( Element convertInSchema);
       Convert   convertOut = new Convert( Element convertOutSchema);

       if (checkContainer(aContainer, is_test))
              {
              loadContainer(aContainer, ExampleService.defaultAnswerValue);
              }
       else
              {

              subscribe.publish(aConteiner, call);
              convertIn.convert(aContainer);
              validate.check(aContainer);

              if (validate.ok())
                     {
                     do(???)
                     }
              else
                     {
                     error.setError(aContainer, exampleService, error, message);
                     }

              subcribe.publish(aContainer, answer);
              }

       convertOut.convert(aContainer);
       return(aContainer);
}
```

Här följer några förtydliganden

Kommunikation

Kommunikation till och från fasaden är inte med i exemplet eftersom det skiljer sig så beroende på kommunikationssätt. Denna del skall hanteras utanför exemplet.

Identifiering / behörighet

Identifiering av anropande person / system sker i kommunikationsdelen. Hur detta skall hanteras inklusive kontroll av behörighet är inte med i exemplet.

Felhantering

Felhanteringen skall leverera information till flera parter,

- **Anropande** – den som anropar en tjänst vill inte veta av exakt vad som är fel, utan mer; beror felet på mig eller Dig ? om jag gjorde fel, vad kan jag göra åt det ? om Du gjorde fel, är det allvarligt, kan jag sända om ? I Appendix E: beskrivs detta mer i detalj och där finner Du också en dokumentation på de ca 20 felkoder som Schenker har definierat skall kunna returneras från en tjänst. Svaret är uppbyggt så att det kan hanteras både av en applikation som av en slutanvändare.

- **Systemadministratören** – Han behöver dels den korrekta interna felkoden, ex från databasmotorn, dels den informationsmängd som funktionen hanterade när felet uppstod för att kunna återskapa felet
- **Helpdesk** – helpdesk behöver bra information för att snabbt kunna åtgärda allvarliga fel
- **Förvaltaren** – Han behöver bra information kring felen för att kunna ta ställning till om justeringar behöver göras i systemet

Felhanteringen sköts om av en generell klass/funktion. Den ser till att den returnerande Containern innehåller bra information till det anropande systemet och det loggas bra information tillsammans med den Container som systemet hanterade i den stund felet uppstod för att stödja de övriga på bästa sätt.

Ett problem som varje utvecklare står inför, det är när han skall ta hand om ett fel. Hur skall han handhava och vad för relevant information skall han sända till helpdesk, systemadministratören och slutanvändaren. Den problematiken sköter denna felhantering åt honom, genom att felhanteringsklassen tillsammans med en Container levererar intressant information till alla dessa parter.

Validering

Validering av Containern sker med en generell klass/funktion med något regelverk, ex DTD eller XMLSchema. All validering kan givetvis inte ske på detta vis men däremot struktur, typning, obligatoriska fält vilket är till en stor hjälp.

Prenumeration / Loggning

Inkommande och avgående Container skickas till en prenumerationstjänst. Därmed kan trafiken avlyssnas och den avlyssningen kan sättas av och på under drift.

Vill man enbart logga händelsen, så ber man en generell loggningsfunktion prenumerera händelsen. Denna loggningsfunktion kan antingen logga hela meddelandet eller endast tidpunkten för händelsen. Det ger bra information om hur många gånger tjänsten anropas, när på dygnet den anropas och loggas tiderna mellan anrop och svar ges också den tidsåtgång som tjänsten tar på sig.

Trots att en Container kan innehålla en mycket stor mängd information av olika karaktär, så är det enkelt att ta fram en generell visning av loggarna och därmed erbjuda systemadministratörerna ett enda verktyg för alla felloggar baserat på Container-konceptet.

Prenumerationstjänsten erbjuder också en möjlighet att återanvända både själva händelsen och information. Tänk Dig följande scenario, Du har ett faktureringssystem som levererar information till en kundreskontra. Du vill ha bra statistik utifrån Din fakturering men reskontran ger Dig endast en rudimentär sådan eftersom den är bara tänkt att stödja reskontraverksamheten. Tänk om det hade gått att prenumerera på informationsmängden som faktureringssystemet levererar, och sända det till en egenutvecklad statistikrutin som ger det stöd Du vill ha istället ??? Denna prenumerationstjänst erbjuder Dig att i framtiden komma åt informationen.

Test

För den som skall anropa en tjänst, behövs en testfunktion som kan användas utan att det påverkar produktionsdata. Det sker genom att det i Containern sänds med en variabel som informerar att detta enbart är en test.

Denna funktionalitet använder vi också när vi idag bygger nya system. Vi börjar tidigt att definiera gränssnitten för de tjänster vårt system skall erbjuda. Vi tar fram ett skalgränssnitt som tar emot en förfrågan och returnerar ett statiskt men korrekt svarsmeddelande. Detta ger främst våra grafiker möjligheten att per omgående kunna bygga visualiseringen tillsammans med slutanvändaren utan att de behöver vänta på att arkitektur och system är färdigbyggda. Detta har visat sig vara en lysande lösning, för den hjälper också till att få slutanvändare och intressenter att bättre förstå det system som skall byggas och de blir bättre på att ställa rätt krav på det framtida systemet.

Konvertering

Det bör finnas en konverteringsrutin både i början och sist i fasaden.

Först för att göra två saker (vanligtvis skilt ifrån varandra). Främst gäller att konvertera mellan de/det format som meddelandet har vid anropet och det interna format och namnformer som systemet använder, ex från XML/HQF till DOM. Sekundärt är att kunna göra eventuella förändringar som behövs, ex det har skett ett namnbyte på någon variabel. Kan också användas för att filtrera bort data eller att lägga till default-värden. Denna funktion används vanligtvis inte utan finns där främst för att underlätta en framtida förvaltningsbarhet, förr eller senare kommer behovet att påverka utseendet och finns funktionen där redan på plats så kan den kompletteras utan att källkoden behöver ändras och därmed undvika att systemet tas ur drift för förändring.

Sist för att dels konvertera mellan det inre och yttre formatet och dels kunna erbjuda olika svarsalternativ, förutom att kunna leverera i samma format som anropet var i, så kan ex HTML, WAP levereras.

Målbild

För att kunna bygga en bra arkitektur förberedd för framtida förändringar, krävs en grundsyn över helheten som är oberoende av plattform, utvecklingsparadigm eller programspråk.

Holism

Alla system skall kunna samverka, både befintliga och nya, på ett enhetligt sätt.

Återanvändning

En hel eller delar av en applikation skall kunna fungera som tjänst. En tjänst skall kunna anropas av flera applikationer eller olika typer av klienter. Befintliga system skall kunna återanvändas.

Återanvändning gäller även för den information som finns och hanteras i systemen.

Fungera på och mot alla plattformar

Fungerar på MVS, AS/400, UNIX, NT, med flera.

Teknik- och implementationsoberoende

Det skall gå att implementera teknik- och programspråksoberoende. Försök implementera detta så långt det går i oberoende tekniker som XML, UML, HTTP, TCP/IP, med flera.

Visualisering på flera olika sätt

– Mot en browser med HTML
– Mot andra applikationer med ex XML
– Mot e-post, skrivare, fax, ...

Språkoberoende

Hantera flera mänskliga språk och programmeringsspråk, både funktionella och objektorienterade.

Integrering

Det skall gå att integrera och skikta system utan att det skapas framtida uppdateringsproblem

Hög förändringsbarhet

Det skall gå att förändra system med ett minimum av ändringar och påverkan. De skall ha god förberedelse inför förändrade krav.

God felhantering

Systemen skall ha en felhantering som ger bra stöd till slutanvändaren, helpdesk, systemadministratören och förvaltaren

Scenarior

Här följer ett antal scenarior kring förändringar som ett system kan utsättas för under sin livstid

FÖRÄNDRAD FÄLTLÄNGD

Systemet är ett Order / Lager / Faktureringssystem byggt som ett Client/server-system med tjock klient. I detta system finns artiklar med ett artikelnummer definierat som 13 tecken alfanumeriskt. Det skall nu förändras till att klara 14 tecken istället, dvs bli en ynka Byte större.

För många system är detta ett av de värsta förändringsscenarierna, att ett id skall bli större eller att byta fälttyp eftersom de flesta system vanligtvis är byggda exakt efter den typ och längd som fälten är definierade till.

Förändringen kommer att kräva:
- Att ett flertal tabeller i databasen måste rekonstrueras
- Att alla delar, i alla skikt, i alla tre systemen måste ändras som hanteras artikelnumret
- Att visualiseringen mot skärm, rapporter mfl måste korrigeras

Det leder till:
- Att stora testinsatser måste göras eftersom förändringarna påverkar så stor del av systemen
- Att databasen måste stängas pga rekonstruktion, dvs systemet står still ett tag
- Att databasen, servern och klienterna måste uppdateras samtidigt med en stor distributionsinsats som följd
- Att kostnaderna blir enorma i förhållande till den ”lilla” förändringen, dvs utöka ett fält med en Byte

Alternativ

Tänk om systemarkitekten hade gjort följande reflektion innan systemet byggdes.

Jag vet ingenting om det framtida behovet av fältlängden på artikelnumret men av praktiska skäl sätter jag gränsen till 255 tecken men tillåter idag inte inmatning eller visualisering av dem med annan storlek än 13 tecken.

Jag skapar en traditionell databas men definierar fältet i databasen som Varchar(255) istället för Char(13). Jag använder Container-konceptet genomgående som bärare av informationen.

Förändringen kommer att kräva:
- Att visualiseringen mot skärm, rapporter mfl måste korrigeras

Det leder till:
- Att databasen inte påverkas alls
- Att små testinsatser måste göras eftersom förändringarna endast påverkar visualiseringen
- Att klienterna kan uppdateras i den takt som behovet finns för utökningen
- Att kostnaderna blir ringa i förhållande till den ”lilla” förändringen, dvs utöka ett fält med en Byte

NY FUNKTIONALITET

Tänk Dig ett informationssystem som levererar information eller funktionalitet till andra system, det kan tex vara ett aktiehandelssystem eller ett transportsystem. Systemet har många externa applikationer som kunder.

Nu vill en av säljarna på företaget sälja denna funktionalitet till en ny stor kund, men den nya kunden vill ha några smärre förändringar och eftersom säljaren så gärna vill ha den nya kunden, så bestämmer man sig för att godkänna förändringarna.

För många client/serversystem är detta en av de värsta förändringsscenarierna, att servern måste förändra den informationsmängd som kommuniceras mellan klienten och server. I detta fall är det särdeles illa, eftersom klienterna finns utanför den egna domänen, ute hos Dina kunder.

Förändringen kommer att kräva:

- Att gränssnittet görs om
- Att alla klientapplikationerna förändras för att kunna hantera det nya gränssnittet

Det leder till:

- Att Du måste be alla Dina kunder förändra sina system så att de kan hantera det nya gränssnittet
- Att Din server och alla Dina kunders system måste uppdateras samtidigt
- Att kostnaderna blir enorma i förhållande till den ”lilla” förändringen, dvs utöka lite funktionalitet för en ny kund.
- Att Dina kunder inte uppskattar förändringen. Även om de vill ha den nya funktionaliteten så kommer de inte gilla tidtabellen att behöva synka sina förändringar med Dig.

Alternativ

Tänk om systemarkitekten hade gjort följande reflektion innan systemet byggdes.

Jag bygger mina gränssnitt enligt Container-konceptet.

Förändringen kommer att kräva:

- Att gränssnittet kompletteras
- Att endast de klientapplikationer förändras som vill hantera den nya funktionaliteten

Det leder till:

- Att Du kan informera alla Dina kunder om den nya funktionaliteten
- Att Din server och alla Dina kunders system kan uppdateras oberoende av varandra
- Att kostnaderna blir ringa i förhållande till den ”lilla” förändringen, dvs utöka lite funktionalitet för en ny kund.
- Att Dina kunder kommer att fortsätta att gilla Dig oavsett om de vill ha förändringen eller ej.

Tänkvärt

Container-konceptet tar bort det synkroniseringsproblem som normalt existerar vid uppdateringar i ett client/server-system.

Det tar också bort behovet att ändra och uppdatera de klienter som inte egentligen påverkas av den nya funktionaliteten.

NY VISUALISERING

Systemet är en webbapplikation, skrivet i något av scriptspråken (ASP, JSP, Perl) och har visualiseringen blandad med funktionalitet. Företaget vill nu två saker, dels ha en helt ny visuell layout i HTML, dels kunna köra applikationen via en WAP-telefon.

Ett system som har blandat visualiseringen och funktionalitet brukar vara svåra att underhålla, svåra att följa logiken i och större förändringar i layouten påverkar logiken.

Förändringen kommer att kräva:
– Att programmet skrivs om för att hantera den nya layouten
– Ett nytt program skrivs för att hantera WAP-telefonen

Det leder till:
– Att det måste till en programmerare att bygga systemen
– Att test av de nya programmen måste göras, både logik och layout
– Att det blir två system framöver att underhålla
– Att kostnaderna blir enorma i förhållande till den ”lilla” förändringen, dvs utöka och förändra visualiseringen

Alternativ

Tänk om systemet var uppbyggt enligt tjänstekonceptet istället.

Förändringen kommer att kräva:
– Att beskrivningen för HTML-layouten skrivs om
– Att en ny beskrivning för WAP skrivs

Det leder till:
– Att en grafiker (ickeprogrammerare) kan göra förändringarna och tilläggen
– Att layouten måste testas men test av logiken behöver ej göras
– Att kostnaderna blir ringa i förhållande till den ”lilla” förändringen
– Att det fortfarande är bara ett system att underhålla

Tänkvärt

I webbvärlden förändras layouten betydligt oftare än logiken bakom, därför måste logik och visualisering helt vara separerade från varandra, vilket tjänste-konceptet erbjuder.

Det blir också allt vanligare att en tjänst skall kunna visualiseras på ett flertal sätt, dels med olika tekniker men också med olika layouter, ex kundspecifika.

KOMMUNIKATION MOT BEFINTLIGT SYSTEM

Ett system behöver hämta statusinformation ur ett befintligt transportsystem. Se liknande exempel i appendix A:Schenker CTTS. Transportsystemet erbjuder idag ingen färdig åtkomstmöjlighet. Därför beslutas att det nya systemet skall accessa transportsystemets databas direkt, eftersom datat skall inte påverkas och det behövs ringa logik för att kunna hantera och tolka informationen.

Efter ett tag beslutas att ytterligare ett nytt system skall hämta information från transportsystemet och eftersom det fortfarande inte finns någon färdig accessväg så beslutas att även detta system skall accessa transportsystemets databas direkt.

Transportsystemet skall nu göra en smärre förändring i sin databas (eller kanske helt bytas ut, effekten blir densamma). Problemet är nu att det är ytterligare två system som påverkas av denna förändring. Däremot så tillkommer det - utifrån de två sidosystemen sett - ingen ny funktionalitet. Dessutom är det ofta så att de som förvaltar transportsystemet känner inte till de övriga systemen.

Förändringen kommer att kräva:
– Att de två sidosystemen också ändras
– Att de två sidosystemen är kända

Det leder till:
– Att de två sidosystemen innefattas i förändringsarbetet
– Att test måste ske av alla tre systemen
– Att systemen måste installeras om samtidigt som databasen förändras
– Att kostnaderna blir enorma i förhållande till den ”lilla” databasförändringen

Alternativ

Tänk om man i det första läget valt ett bygga en fasad som en tjänst mot transportsystemet istället.

Förändringen kommer att kräva utifrån de två sidosystemen:
– Ingenting

Det leder till:
– Att kostnaderna blir inga i förhållande till den ”lilla” databasförändringen

Tänkvärt

Det är förståeligt att system byggs på så sätt att de accessar andra systems databaser direkt men samtidigt så skapas ett framtida underhållsproblem. Värsta scenariot är att förändringen kan inte genomföras över huvud taget pga att den påverkar andra system och det går inte att hantera det synkroniseringsproblem som uppstår, ex att det går inte att ta ner systemen samtidigt.

Som exemplet i appendix A visar, så tar det inte speciellt lång (mer) tid att ta fram en tjänst mot databasen istället och därmed undvika det framtida underhållsproblemet. Eftersom bla Ciel är ett standardsystem där Schenker har liten påverkan, så innebär den installerade tjänsten att systemet Ciel kan fortsätta att utvecklas och underhållas av sin ägare (DCS Group) utan att det påverkar de system som använder tjänsten.

Kvalitetsattribut

Tjänstekonceptet baserat på Containerkonceptet har följande påverkan på dessa kvalitetsattribut:

Prestanda

Anrop som görs inom ett program påverkar normalt endast stacken, vilket har mycket liten påverkan på prestandan, under förutsättning att med anropet skickas endast en referens till Containern. Alla andra anrop innebär start av kommunikation, upp och ned-packning av data, felhantering för anropet mm, vilket givetvis påverkar prestandan mycket negativt. Detta problem har alla lösningar som arbetar med distribuerade system, oavsett J2EE mfl eller de beskrivna koncepten.

Storleken på det serialiserade meddelandet kan också ha negativ påverkan. Väljs ett format som XML så blir meddelandet stort, väljs ett kompaktare format, ala MARC, så kan det till och med bli mindre än traditionella serialiseringsformer eftersom inga tomma fält, högerblanka eller nollutfyllnader finns med i meddelandet.

Generellt gäller att man skall se upp med att få för många anrop vid ett tillfälle som innebär trafik över nätet.

Skalbarhet

Skalbarheten har samma bekymmer som prestandan, har ett system många upp och nedpackningar med trafik över nätet, så får det stor påverkan på CPU-belastningen. Däremot så erbjuder tjänstekonceptet att tjänsterna kan spridas ut på flera maskiner för att därmed förbättra skalbarheten. En tjänst kan också existera i multipler över flera maskiner, låt ex en proxy sprida anropen.

Container-konceptet skalar bra eftersom en Container med lätthet kan hantera stora informationsmängder och påverkas i princip inte alls av att meddelandena blir större. Däremot kan valet av serialiseringsformat få stor påverkan, på samma sätt som för prestandan.

Eftersom Container-konceptet hanterar datat som en enhet, så kan det, vid behov, med fördel komprimeras, ex med LZ77 komprimering.

Säkerhet

Ett taggat meddelande med fältnamn och värden i klartext kan bli en risk för säkerhetsklassad information. Eftersom Containern hanterar data som en enhet, så kan den både krypteras och få en digital signatur för att öka säkerheten.

För distribuerade system så är säkerheten alltid ett problem med säkerhetsklassad information eftersom både identifiering och behörighet måste verifieras på flertal ställen.

Prenumerationstjänsten kan bli ett problem, då den lyssnar av någon annans trafik.

Tillgänglighet

Ett flertal funktioner kan förändras under drift utan att systemet behöver tas ner. Det gäller

- **prenumeration** – prenumerationstjänsten skall vara byggd så att den kan slå på och stänga av prenumerationen under drift, även lägga till och ta bort.
- **loggning** – använder sig av prenumerationstjänsten och kan därmed slås på och av efter behov under drift
- **meddelandekonverteringarna** – regelverken för dem hämtas från fil eller databas

Om en tjänst är dubblerad och anropas via en proxy, så kan den ena applikationen tas ner för underhåll utan att tjänsten stängs. Om en tjänst stängs, så skall en speciell returkod levereras så att den som anropar blir varse på rätt sätt, exempel på lösning visas i appendix E: Felhantering

Funktionalitet

Den funktionalitet som ges är främst den som tjänstens fasad erbjuder:

- Kommunikation
- Identifiering
- Behörighet
- Validering
- Konvertering
- Loggning
- Felhantering
- Spårbarhet
- Prenumeration
- Visualisering
- Test
- Modifierbarhet

Efter att arbetat med Containers ett tag, så byggs det upp ett stort generellt funktionsbibliotek för dess hantering som kan användas var helst en Container används.

Användbarhet

Fasaden som byggs blir nästan identisk för varje implementerad tjänst. Det gör att byggandet av en fasad sker med ett antal hjälpklasser och en programmall och kan byggas på relativt mycket kort tid. De som bygger den bakomliggande funktionaliteten slipper också att ta hand om all den funktionalitet som fasaden erbjuder.

Vid byggandet av en tjänst, så bygger vi idag det första vi gör, en testfunktion som levererar tillbaka en statisk men ett korrekt svar. Det gör att främst våra grafiker kan påbörja byggandet av handhavandet av systemet utan att all funktionalitet finns på plats. Detta gäller givetvis även de applikationsutvecklare som vill nyttja en tjänst som ännu inte finns färdig.

Modifierbarhet

En av kärnidéerna med Container-konceptet är att det skall stödja modifierbarhet. Det skall gå att ändra och lägga till informationsmängder utan att det får den lämmeltågseffekt som många client/server-system drabbas av.

Även tjänstekonceptet förstärker detta genom att fungera som ett skyddande fönster mot de bakomliggande funktionerna.

Koncepten är så starka inom detta område att många förändringar som görs kan ske utan att det påverkar alls de delar som inte har behov av att bli påverkade.

Portabilitet

Koncepten är i grunden plattformsoberoende och implementationsoberoende. Implementeras koncepten med oberoende och portabla tekniker, såsom XML, TCP/IP, HTTP, Java etc, så blir helheten mycket portabel.

Även om implementationen inte blir med portabla tekniker, så blir ändå systemen relativt enkla att flytta pga att kärnidén är portabel.

Återanvändbarhet

Återanvändbarheten finns i flera dimensioner:

– **System** – befintliga system kan återanvändas helt eller i delar genom att de ges en fasad. Denna lösning används mycket för att ge systemen en ny visualisering, ex med browsers
– **Information** – prenumerationstjänsten ger möjlighet att komma åt färsk information.
– **Kodbibliotek** – Fasadens uppbyggnad och funktionerna kring Containern kan återanvändas eftersom de är helt generella.
– **Mönster** – Många delar i systemen kommer att hanteras på samma sätt vilket leder till ökad kunskap om hur de skall byggas men också – inte minst - hur systemen kommer att uppföra sig.

Integrerbarhet

Tjänstekonceptet är ett starkt koncept för just integrerbarhet. Det ger en bra skiktning mellan system och ger en bra kontroll över hur systemen integrerar med varann.

Den vanligaste integreringsformen och samtidigt den ur underhållssynpunkt den värsta är att från ett system anropa ett annat systems databas direkt. Med en relativt liten insats kan denna ersättas med anrop via en tjänst och därmed få en betydligt trevligare framtida förvaltning.

Testbarhet

En tjänst skall erbjuda följande grundläggande tester,

– **Kommunikation** – visa att kommunikationen fungerar och att tjänsten är igång
– **Informationskontroll** – Det anropande meddelandet skall valideras och kontrolleras utan att det (helst) påverkar produktionsdata.

Loggningen via prenumerationstjänsten erbjuder kontroll av pågående trafik.

Ytterligare en funktionalitet är att behövs det läggas in debug-satser, så gör Containern det enkelt att spara undan all den data som bearbetas och som kan vara intressant att studera.

Tid till marknaden

Hur snabba koncepten är beror lite på var man är i utvecklingskedjan. När man precis skall börja utveckla med dessa koncept, så tar det initialt längre tid, beroende på att det saknas grundläggande funktionalitet som måste tas fram och utvecklarna är ovana. Allt eftersom tiden går så återanvänts de mönster, kunskap och källkodsbibliotek samt, inte minst, de tjänster som nu är framtagna.

Konceptens styrka kommer till sin rätt när system skall vidareutvecklas och det skall kompletteras med ny funktionalitet, ex kan ny funktionalitet läggas till en tjänst för att stödja en ny kund utan att de befintliga kundernas system måste förändras.

Kostnader

Precis som för 'tid till marknaden' så är kostnaderna för koncepten initiala. De får många gånger version 1 av ett system att bli något dyrare. Konceptens styrka kommer i de följande versionerna och det leder till förenklad förvaltning och lägre livstidskostnader.

Slutsatser /Diskussion

Container-konceptet och tjänstekonceptet har idag använts i flera olika system, i flera olika miljöer med mycket goda erfarenheter. Schenker [Schenker] har idag valt detta sätt att bygga i sina utvecklingsregler och har idag 14 tjänster framtagna på MVS, AS/400, Unix, NT . Ett av Ementors produkter inom Content Management System, Econgero [Econgero], är idag helt uppbyggd på detta sätt med XML som bas. I nästa version kommer också fullt stöd för Web Services fast med de förändringar som tidigare nämnts i detta dokument.

FÖRDELAR

Jag tycker att koncepten genom åren har visat att de håller vad de lovar, de ger en hel del funktionalitet som uppskattas under systemens livslängd, de ger betydligt mindre underhåll och påverkan än traditionellt byggda system.

För bibliotekssystemet Libra [Libra] och CMS-systemet Econgero så är dessa koncept en del av dess framgångar. Systemen är väldigt flexibla och modifieringsbara vilket leder till att trots att det har många kunder som var och en önskar sina tillägg så finns det bara en gemensam version var av systemen.

Rent tekniskt har det inte heller varit några stora problem att hantera det här. Mot MVS CICS valde vi något tillfälle att lägga fasaden på en Unix-maskin som i sin tur kommunicerade med CICS-socket och DB2-connect.

NACKDELAR

Om man bortser från de nackdelar som nämns i dokumentet, prestanda och meddelandestorlek, så har det tekniskt inte några påtagliga nackdelar. Istället hittar man dem på ett helt annat ställe, på det mentala planet.

När en programmerare börjar sin karriär, så bygger han i första hand funktioner för att klara de funktionella krav som ställs på systemet. Fjärran är tankarna på att systemet skall leva lång tid, komma i fler än en version etc. Det leder till att systemet blir oflexibelt och får bla stark kontroll och typning. De får ingen träning i att bygga modifieringsbara system, systembaserade på lösa skikt.

Det saknas också en debatt kring ämnen som förvaltningsbarhet, modifierbarhet, heterogena system, blanda gammalt och nytt.

Det leder till att det saknas fokus på att lösa de problem som dessa ämnen har. Saknas fokus så kommer det heller inte att lösas på egen hand (man ser endast det man fokuserar på). Java-programmerare fokuserar på Java Community, Microsoft-utvecklare fokuserar på nya .Net community, och ser därmed enbart det som dessa communities diskuterar.

De flesta databaser bygger på en tvådimensionell begreppsvärld, post och fält. Det är egentligen först med XML som utvecklare börjar tänka om data som trädstrukturer istället.

Projekten är också fokuserade på att leverera funktionalitet så fort som möjligt till så låg kostnad som möjligt, det är sällan livstidskostnader diskuteras när ett system byggs i version ett. Projektledaren är därmed inte heller nämnvärt intresserad av ovan nämnda ämnen eftersom de tar mer tid initialt och därmed ökade kostnader för projektet.

FRAMTID

Det är lättare idag att diskutera dessa koncept än tidigare, tack vare XMLs genombrott och inte minst den debatt kring Web Services som finns idag.

Många större företags IT-chefer och stadsplanerare har visat stort intresse för koncepten eftersom de idag sitter med en komplex arkitektur som de har svårt att hantera och attraheras av dessa koncept eftersom den kan smygas in arkitekturen och hanterar många av de problem som de har behov av att få lösta.

Den process som pågår idag med webifiering, Web Services, distribuerade system mm gör att övriga världens fokus sakta men säkert närmar sig det som de beskrivna koncepten vill uppnå.

För Dig som blir attraherad av dessa tankar, vill jag önska lycka till, för de är verkligen bra. När vi utvecklade bibliotekssystemet en gång i tiden så fick vi en aha-upplevelse nästan varje dag under lång tid över vad Containern kunde lösa åt oss och de synergieffekter vi fick.

Men man får dem inte med automatik. Eftersom det inte finns någon erfarenhet, ingen debatt, ingen community som jobbar med dessa frågor, så är det svårt att få omgivningen att i verkligheten arbeta med, även om personerna i fråga tycker som vi.

Vill Du införa detta så måste Du vara beredd på att både vara Mentor och polis, att utbilda, informera och kontrollera, hela tiden. Därför att utvecklarna har så lätt för att återgå till det gamla sättet att bygga system.

Referenser

[Libra] Axiell bibliotekssystem med produkten Libra. Läs mer på www.axiell.se

[Lundgren 2001] Stefan Lundgren. Meddelandehantering, projektarbete för IT-arkitekt 9. Beskriver ett sätt att beskriva små, kompakta meddelande för aktiehandel med krav på flexibilitet och hög prestanda.

[Schenker] Kontaktperson Henryk Zienkiewicz, hz@schenker.com.

[DCS Group] Säljer transportsystemet Ciel, se mer på http://www.dcstrans.com/

[EDIFACT] Förenta nationerna har definierat EDIFACT, med både syntax och semantik för utbyte av EDI mellan företag

[Econgero] Ementors Content Management System Econgero, läs mer på www.econgero.com

[UDDI] UDDI, Universal Description, Discovery, and Integration, läs mer på www.uddi.org

[J2EE] Java 2 Platform Enterprise Edition, se http://java.sun.com/j2ee/

[.NET] Microsofts nya plattform .NET, se mer på http://www.microsoft.com/net/

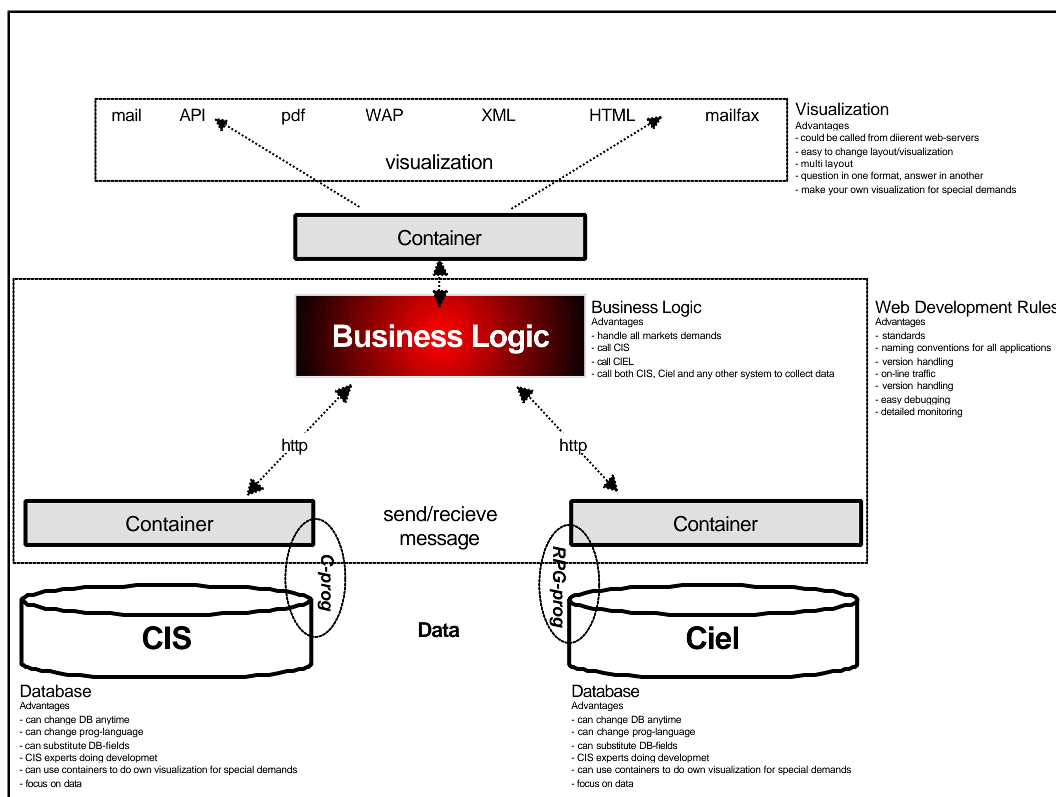
[Buschmann] Frank Buschmann et. al. Pattern-oriented software architecture.

[W3C] World Wide Web Consortium, www.w3c.org

Appendix

A: Exempel Schenker CTTS

Här kommer ett exempel ur verkligheten, Schenkers Common Tracking and Tracing System (CTTS) [Schenker] vars gränssnitt är uppbyggda som tjänster med en fasaduppbyggnad. De använder Containerkonceptet och därmed kan visualiseras på samma sätt oavsett om det är en applikation som anropar eller en browser, wap-telefon etc. I exemplet visas arkitekturmönstret för CTTS, hur meddelandena (Containrarna) ser ut samt i vilka miljöer det är implementerat på.



Schenker har idag ett flertal system som hanterar godsflöden världen över. Det är också olika system för Inrikes, Utrikes samt Sjö och flyg. I samband med webbens intåg och att Schenker börja låta sina kunder accessa de interna systemen, har det uppstått ett behov att få alla dessa system att upplevas som ett enda.

Varje system erbjuder en tjänst där förfrågningar på gods kan göras. Uppbyggnaden är den samma för allihop. Däremot så erbjuder de olika mycket information, beroende på om informationen hanteras i systemet eller ej. Därför erbjuder systemen CIS och Ciel olika mycket funktionalitet och returnerar olika informationsmängder, ex från CIS returneras endast händelser som har skett med ett gods medan Ciel kan också returnera de händelser som godset förväntas genomföra.

Det här är inga problem att hantera tack vare Container-konceptet, eftersom det tillåter att information både finns och saknas. Ett problemområde med systemet är att identiteter

som kundnummer och godsnummer endast är unika i varje enskilt system men är det inte inom Schenker. Även om ”Business Logic” vet en hel del om de bakomliggande systemen, så kan den ibland bli tvingad att anropa flera bakomliggande system parallellt för att söka och hämta information.

Business Logic

Det är egentligen är för starkt ord för detta system. Den gör följande:

- Tar emot en förfrågan och validerar den
- Beräknar vilket/vilka av de bakomliggande systemen som kan tänkas ha information
- Anropar systemet/systemen. OBS sker med exakt samma fråga som inkommande fråga (container). Trådas upp och sker parallellt.
- Sammanställer (Gör en merge om fler svar kommer)
- Filtrerar (tar bort information som systemägaren inte vill visa)
- Visualiserar (med XSLT) eller returnerar datat till anropande system

CIS

Transportsystem för utrikes trafik i bla Tyskland. Körs i Unix och är byggt med JAM, ett C-liknande 4GL-språk. Utvecklat för Schenker Tyskland av Schenker.

Ciel

Transportsystem för utrikes trafik i bla Sverige. Utvecklas av DCS Group [DCS Group] i England och är ett standardpaket som används om många stora företag. Körs i AS/400 och är skrivet i RPG. DCS kom i kontakt med Containerkonceptet för drygt 2 år sedan, eftersom Schenker krävde att deras system måste ha detta gränssnitt. Idag erbjuder DCS i Ciel ett gränssnitt uppbyggt enligt det Containerkoncept och Tjänstekoncept som är beskrivet i detta dokument till alla sina kunder.

Uppbyggnad

Hur lång tid tar det att bygga ett tjänstekoncept som är beskrivet här ?
För både CIS och Ciel tog det lika lång tid och skedde på följande vis:

- Jag anländer till Essen/Leeds ca kl 11:00.
- Kort presentation och genomgång av företag mm, lunch
- Utbildar och förklarar Containerkonceptet för chefer och utvecklare 3-4 timmar
- Därefter påbörjas utvecklingen av deras utvecklare med liten paus för middag och sömn.
- När jag åker nästa dag kl 14:00 så är version 1.0 redan testad och klar

Tjänsterna har sedan reviderats och förbättrats en hel del men jag vill ända visa att det går på relativt kort tid att få upp en hyggligt fungerande tjänst, oavsett plattform. En hel del tid tog det faktum att båda utvecklingsteamen var nya på web-tekniken vid denna tidpunkt. Det skall väl också poängteras att gränssnittsdokumentationen var klar innan mötet.

Övrigt

Det framgår inte av bilden, men gränssnitten för CIS och Ciel kan anropas direkt, anropet måste inte gå via Business Logic. De kunder/anställda som anropar systemet direkt, får mer funktionalitet och returnerande information, efter systemen erbjuder detta. Business Logic erbjuder endast ett fåtal tjänster och filtrerar dessutom en del av informationen.

Det som är värt att notera, är att det är samma tjänst som anropas och det är samma svarsmängd, oavsett om det Business Logic som anropar eller något annat anropande system.

Fasadens uppbyggnad är densamma för Business Logic, CIS och Ciel, de anropas på samma sätt. Viktigt är att alla fasader kan leverera svaret i ett dataformat, även för ett skikt som egentligen endast är tänkt att visualisera. Därmed kan Schenker erbjuda en ytterligare övergripande tjänst, som anropar Business Logic som en komponent i ett större system.

Meddelandeexempel

Här visas ett enkelt exempel på hur ett anrop och ett svar kan se ut. Formatet i exemplen är HQF (HTTP QueryString Format) och XML. Mer om de olika formaten kan läsas under Appendix D: serialiseringsformat.

Kort intro om skillnaden mellan HQF och XML. I ett större meddelande så blir storleken ungefär densamma för båda formaten. I HQF skiljs nivåer åt med en punkt medan i XML med att en tagg definieras inom en annan tagg. Ex

– HQF: request.service.action=select
– XML: <request><service><action>select</action></service></request>

OBS Eftersom hela strukturen finns representerad i variabelnamnen i formatet HQF, så finns det inget krav på någon alls ordning mellan variablerna i meddelandet. De kan komma i vilken ordning som helst.

Se även information under appendix C: Namnformer

Hämta sändningshändelser för ett specifikt gods (HQF)

```
request.select.transport_document_number=1111111116
request.format.mime=text/html
request.service.action=select
request.service.method=waybill
```

Svar

```
response.service.name=tracking
response.service.version=1.0
system.error.id=0
data.contract_condition_code=PAR
data. transport_document_number=1111111116
data.departure_place_name=UDDEVALLA
data.destination_place_name=STOCKHOLM
data.acceptance_date=20011224
data.status_event_list.0.status_event_code=1
data.status_event_list.0.status_event_place=UDDEVALLA
data.status_event_list.0.status_event_date=20011224
data.status_event_list.1.status_event_code=4
data.status_event_list.1.status_event_place=GÖTEBORG
data.status_event_list.1.status_event_date=20011224
data.status_event_list.2.status_event_code=1
data.status_event_list.2.status_event_place=GÖTEBORG
data.status_event_list.2.status_event_date=20011225
data.status_event_list.3.status_event_code=4
data.status_event_list.3.status_event_place=STOCKHOLM
data.status_event_list.3.status_event_date=20011225
```

Hämta sändningshändelser för ett specifikt gods (XML)


```
<xml>
  <request>
    <select>
      <transport_document_number>1111111116</transport_document_number>
    </select>
    <format>
      <mime>text/html</mime>
    </format>
    <service>
      <action>select</action>
      <method>waybill</method>
    </service>
  </request>
</xml>
```


Svar


```
<xml>
  <response>
    <service>
      <name>tracking</name>
      <version>1.0</version>
    </service>
  </response>
  <system>
    <error>
      <id>0</id>
    </error>
  </system>
  <data>
    <contract_condition_code>PAR</contract_condition_code>
    <transport_document_number>1111111116</transport_document_number>
    <departure_place_name>UDDEVALLA</departure_place_name>
    <destination_place_name>STOCKHOLM</destination_place_name>
    <acceptance_date>20011224</acceptance_date>
    <status_event_list>
      <status_event>
        <status_event_code>1</status_event_code>
        <status_event_place>UDDEVALLA</status_event_place>
        <status_event_date>20011224</status_event_date>
      </status_event>
      <status_event>
        <status_event_code>4</status_event_code>
        <status_event_place>GÖTEBORG</status_event_place>
        <status_event_date>20011224</status_event_date>
      </status_event>
      <status_event>
        <status_event_code>1</status_event_code>
        <status_event_place>GÖTEBORG</status_event_place>
        <status_event_date>20011225</status_event_date>
      </status_event>
      <status_event>
        <status_event_code>4</status_event_code>
        <status_event_place>STOCKHOLM</status_event_place>
        <status_event_date>20011225</status_event_date>
      </status_event>
    </status_event_list>
  </data>
</xml>
```

B: Exempel Bibliotekssystemet LIBRA

Bibliotekssystemet LIBRA [Libra] utvecklades av företaget Biblioteksprogram med start från slutet 80-tal, som ett system för mer krävande och större bibliotek. Dessutom fanns ytterligare en produkt, BIBS, ett enklare bibliotekssystem med en traditionell databas.

Idag drivs LIBRA av företaget Axiell Bibliotekssystem. Axiell köpte våren 2001 upp störste konkurrenten och dess system BTJ 2000. Båda produkterna hade i princip halva svenska marknaden var och därmed har idag Axiell i stort sett hela svenska marknaden för bibliotekssystem.

LIBRA bygger på en biblioteksstandard, MARC (Machine Readable Code, utvecklad under 60-talet) och KRS (Katalogregler för Svenska Bibliotek). Tyvärr finns det ett oändligt antal varianter av MARC men de bygger allihop på samma grundidé.

MARC-formatet var från början tänkt enbart för att visualiseras på katalogkort. Det byggdes inte för att egentligen hanteras i en databas med sökmöjligheter etc. Eftersom MARC har funnits under så lång tid och det har skapats enormt stora databaser med detta format, så har det levt kvar. MARC-formatet har en mycket komplex datastruktur för att beskriva ett media.

För att kunna hantera ett media, ställs vissa grundläggande krav på databasen:

- Varje fält måste kunna ha obegränsad längd (vill någon mata in hela romanen i ett fält, måste systemet klara detta)
- Varje fält måste kunna upprepas i oändligt antal gånger (finns 17 författare skall alla med)
- Databasen måste kunna hantera obegränsat med fält per media (teoretiskt behövs drygt 1000 fält för att kunna beskriva olika media, ett media kan vara en bok med tillhörande CD och nothäften)
- Fält måste kunna grupperas för att bilda en enhet (ex författarens namn, födelseår, födelseort, mm för att kunna skilja en författare från en annan med samma namn). Söksystemet måste kunna söka/sortera på enskilt fält eller en enhet.
- Biblioteket måste kunna få definiera sina egna fält och fältgrupperingar, tillägg tillåts under drift (runtime)
- Ett fält skall kunna ha flera sök/sorteringsfält knutna till sig (ex författarens namn måste registreras som det står i mediet, medan författaren kanske stavas vanligtvis annorlunda i Sverige. I en mycket stor databas hittades 40 olika stavningar på Moamar al-Khadaffi)
- Det skall klara hela latinska alfabetet, kyrilliska, grekiska plus specialtecken såsom matematiska etc
- Det skall gå att söka ett ord oavsett om det är möjligt att stava ordet på ett tangentbord eller inte, dvs kunna söka på Günther utan att bokstaven ü finns på tangentbordet. Sökningen sker genom att användaren tar bort diakriten (punkterna) och anger grundbokstaven istället, dvs Gunther.

För att kunna hantera denna komplexa struktur, uppfanns Containern som en paketering av datat. Containern lagras i databasen oformaterad. Sökindex skapas 'manuellt' ur grundinformationen och konverteras och filtreras innan indexet skapas. Denna metod ger många fördelar, exempel beskrivs lite senare.

MARC-formatet är ett taggat och kompakt format, endast i klartext. Exempel på en skönlitterär bok kan vara:

```
041aswe                       Språk svenska
081cHc.01                     Klassificering
096aHc                        Finns på hylla Hc
100aGuillou, Jan;c1944-       Författare, född 1944-
```

```
245aFiendens fiende             Titel
260aHöganäs;bBra böcker;c1989   Tryck: var, av vem, årtal
```

MARC-formatets uppbyggnad är först en övergripande rubrik för flera fält. Denna är treställig, ex 100 för författare. Det följs att två indikatorer som ger en ytterligare precisering av informationen (är inte med i exemplet). Varje delfält börjar med en enställig kod, ex a, därefter själva informationen följt av en delimiter (semikolon i exemplet).

Som Du ser så blir resultatet väldigt kompakt, eftersom det inte existerar tomma fält, inga högerblanka och det är en mycket kompakt taggning, exemplet ovan blir ca 100 bytes.

Volymuppgifter

Hälften av titlarna i ett folkbibliotek är skönlitterära böcker, klassificerade under H. I snitt så innehåller en post med skönlitterär bok, ca 700 Bytes data inklusive taggningen. Med index och fritextindex blir posten ca 4KB. Ett stort bibliotek har ca 100 tusen skönlitterära titlar vilket blir ca 400 MB data. Facklitteratur och andra media tar vanligtvis mer plats men kontentan är ändå att mediakatalogen tar mycket liten plats, trots sin storlek och komplexitet.

Index

Index skapas manuellt utifrån grunddatat. Fördelen med detta är att datat kan konverteras och filtreras innan det sparas i databasen. Sökvillkor som en användare anger, filtreras och konverteras via samma rutin, därmed blir träffchansen mycket större.

Ett exempel är telefonnummer, det finns ett stort antal sätt att ange telefonnummer, många varianter är personliga. Att få en katalogisör och en låntagare att välja samma format är i det närmaste omöjligt. Exempel på hur en lagring kan vara:

031-895 000	vilket kan lagras som	031895000

Ett sökvillkor på

031 89 50 00	blir också inför sökningen	031895000

vilket leder till att matchning sker trots att grunddatat och sökningen skiljer sig åt.

Ytterligare en variant är att ett sökfält kan existera flera gånger, ett exempel är Sven-Erik. Det är inte lätt att veta hur en författare stavar sitt namn eller om Sven-Erik är ett eller två namn. Detta löses genom att filtreraren skapar två poster istället, SVENERIK och SVEN ERIK. Det ger låntagaren möjlighet att få träff oavsett om de stavar Sven-Erik som ett eller två namn. Hur index hanteras för övrigt i databasen tas inte upp av detta dokument

Hantering

Eftersom en post alltid hanteras som en helhet (en variabel av typen Container) så blir huvudprogrammet mycket enkelt. Koden blir ungefär:

- Hämta en post med id n. Visualisera post med vy x
- Hämta post med id n för editering. Visa post med editeringsvy y. Om OK, spar post

Summa summarum

Hanteringen av Containern togs fram för att kunna hantera komplexa data i ett Cobolprogram. Hjälpfunktionerna skrevs i C. Synergieffekterna blev väldigt positiva, ex det kompakta formatet, enkelheten att administrera det i Cobol, index-filtren, tillåta massor av visualiseringar som säljare eller kund kunde ta fram själva och mycket bra prestanda.

Ytterligare en synergieffekt är att en post uppbyggd som en Container blir mycket modifierbar utan att det krävs några förändringar alls i koden (eventuellt smärre).

C: Namnformer

Vid all kommunikation mellan parter så uppstår två problem som måste hanteras, dels syntaxen (hur saker stavas), dels semantiken (vad saker betyder).

Schenker [Schenker] hanterar detta på följande vis:

Namnform

Alla variabelnamn, fältnamn mm, måste beskrivas enligt följande regelverk (en delmängd):
- Amerikansk stavning i klartext – alla förstår och ger en enhetlig hantering
- Förkortningar tillåts ej, bortsett från vedertagna (id, temp) – viktigt med förståelsen
- Endast gemener (små bokstäver) – En kombination av små och stora bokstäver ställer ofta till problem.
- Orden separeras med ett understrykningstecken – ökar läsbarheten

Namnen skall fungera på många plattformar och på en mängd olika sätt, detta regelverk underlättar kommunikationen mellan plattformar.

Betydelsen

Även om variabelnamnet är i klartext och med tillhörande kommentarer, så kan det ändå uppstå missförstånd med tolkningen av värdet. För de system som kommunicerar med andra system (eller kan tänkas i en framtid) ställs som krav att det dessutom skall refereras till en vedertagen termkatalog. Schenker har valt att i första hand använda Förenta nationernas EDIFACT [EDIFACT].

Exempel från meddelandedefinition svar från en godssökning (delmängd).

Namn	O	Typ	Standard	Förklaring
contract_condition_code		String	TSR 4065-PAR 4065-COM 4065-BUD 4065-CLD 4065-DIR 4065-SPC	Produktnamn: PAR = Parcel, COM = Comfort, BUD = Budget, CLD = Coldsped, DIR = Direct, SPC = Special
transport_document_number		String	RFF 1153-AAS DE 1154	Sändningsnummer, normalt 10 siffror
departure_place_name		String	TDT 3227-5 DE 3224	Avsändningsort
destination_place_name		String	TDT 3227-8 DE 3224	Mottagningsort
acceptance_date		Date	DTM 2005-143 DE 2380	Transportdag
status_event_list		Array		Vektor med sändningshändelser

D: Serialiseringsformat

Schenker använder idag två olika serialiseringsformat i klartext, dels XML, dels HQF (HTTP QueryString Format). Dessa två kommer att beskrivas här. I Bibliotekssystemet LIBRA hanteras ett format som heter MARC, det kommer också att beskrivas lite kort.

Alla dessa format är i klartext och XML och HQF med en ganska generös taggning. Det är fullt möjligt att skapa ett ännu kompaktare och binärt format. Det kommer inte att beskrivas i detta dokument. Stefan Lundgren gjorde ett projektarbete i kurs IT-arkitekt 9 där han beskrev en lösning på ett mycket kompakt format som skulle kunna fungera som bas till en implementering av Container-konceptet.

Vanligtvis behövs flera olika implementationer, ex XML och HQF. Anledningen är att olika plattformar kan inte alls hantera eller är inte så bra på att hantera vissa format. Ett kompaktare format kan behövas för bättre prestanda.

Det viktiga är att de olika implementationerna är kompatibla med varandra i sin uppbyggnad. Det skall gå att bygga en generell funktion som kan konvertera mellan formaten fram och tillbaka.

XML

XML är idag en bra standard att bygga en implementation av Container-konceptet på. Dels för att det har blivit så vedertaget, dels för att det finns en sån stor flora av bra och billiga (gratis) verktyg för att hantera den.

Ett problemområde är att de flesta verktyg och åsikter på marknaden stöder stark typning och kontroll. Container-konceptet vill inte ha kontroll eller typning. Det går att beskriva ett meddelande med DTD men inte fullt ut för att täcka alla utseenden som ett meddelande kan få. Det går att beskriva att en tagg kan innehålla vad som helst (ANY) mm, men beskrivningen blir onödigt krånglig. En ny variant för kontroll av meddelanden är på G, XML Schema. Den har mer möjligheter att beskriva ett meddelande på ett lösare sätt.

Det finns fler fördelar med XML. Hanteringen i programmet sker via DOM, ett standardiserat gränssnitt för programmeraren vilket leder till att programmerarna hanterar meddelandet på ett likartat sätt. Konvertering av ett meddelande sker med XSLT, också det en standard. XSLT är ett kraftfullt verktyg för att konvertera ett XML-meddelande till olika format, ex till ett annat XML-format, HQF, HTML, WAP.

XML har idag blivit vanligt att använda i nyskrivna system för informationsutbyte mellan system. Denna trend kommer att öka ytterligare i takt med att Web Services slår igenom med full kraft.

Nackdelarna med XML är:

- **Storleken** - Ett större meddelande blir ca 10-30 gånger större än ett kompaktare format
- **Prestandan** - Det tar en hel del kraft för att packa upp/ner ett XML-meddelande.
- **Vissa plattformar** – XML finns inte tillgänglig på alla plattformat än eller kan endast hanteras med svårigheter
- **DTD** – Det finns inte fullt stöd för Container-konceptet med DTD:er och med vissa restriktioner med XML Schema

Trots nackdelarna talar fördelarna för att XML idag är ett förstaval vid implementering av Container-konceptet

HTTP QueryString Format (HQF)

Ett problem med HQF är att det inte kan beskriva en trädstruktur i sig själv eftersom det enbart finns en enda nivå på variablerna. För att lösa detta problem, införde Schenker trädstrukturen i namnet genom att införa en punkt som delimiter mellan nivåer.

Exempel

```
HQF: system.error.id=0
XML <system><error><id>0</id></error></system>
```

Exemplet skall uttolkas så att

id är ett fält i objektet/containern error och med värdet noll
error är ett fält i objektet/containern system

Fördelarna med att använda HQF som serialiseringsformat är:

- **Browsers** – HQF är standard för en browser att sända in variabelinformation till en webbserver. Genom att använda namnformer med inbyggda träd så kan ett Container-meddelande enkelt skapas med browsern från ex ett formulär. Det går inte att styra i vilken ordning browsern sänder sin information men det gör inget eftersom Container-konceptet godtar vilken ordning som helst.
- **En vektor** – Många plattformar har idag fortfarande problem att hantera trädstrukturer ala XML. Däremot kan alla hantera en traditionell vektor/tabell bestående av två fält, namn och värde.

För de plattformar som har tillgång till hantering av associativa vektorer (ex HashTables, Dictonary) bör ju helst använda dem, i annat fall en traditionell vektor.

Nackdelarna med HQF är:

- **Storleken** - Ett större meddelande blir ca 10-30 gånger större än ett kompaktare format
- **Prestandan** - Det tar en hel del kraft för att packa upp/ner ett HQF-meddelande.

Om Ditt system kommunicerar med en browser, bör HQF vara minst ett av Dina implementationer av Container-konceptet, likaså om Du använder plattformar som inte kan hantera XML.

MARC

MARC-formatet är ett specifikt format för bibliotekssystem men finns med här mera som ett förslag på hur ett kompaktare format kan se ut.

MARC-formatet är ett taggat och kompakt format, endast i klartext. Exempel på en skönlitterär bok kan vara:

```
041aswe                           Språk svenska
081cHc.01                         Klassificering
096aHc                            Finns på hylla Hc
100aGuillou, Jan;c1944-           Författare, född 1944-
245aFiendens fiende               Titel
260aHöganäs;bBra böcker;c1989     Tryck: var, av vem, årtal
```

MARC-formatets uppbyggnad är :
Huvudfält - först en övergripande rubrik för flera delfält. Denna är treställig, ex 100 för författare.
Två indikatorer – dessa ger en ytterligare precisering av informationen (är inte med i exemplet). Exempel är att beskriva namnformen för en författare. Är registreringen som ett enkelt släktnamn (Bergström, Martin) eller som namn i rak följd (Martin Bergström)

Delfält – Varje delfält börjar med en enställig kod, ex a, därefter själva informationen följt av en delimiter (semikolon i exemplet).

MARC-formatet tillåter att både huvudfälten och delfälten upprepas otal gånger.

Diakriter
Som kuriosa kan här beskrivas hanteringen av latinska tecken med diakriter (prickar, ringar, komma etc), grekiska tecken etc. För att denna lösning skall kunna hanteras, måste några kriterier vara uppfyllda:

- **Variabel fältlängd** – Hanteringen gör att diakritiska bokstäver tar mer plats, därför behöver fältlängden vara variabel så att informationen får plats efter expansion
- **Manuella index** – Det går inte att använda den inbyggda indexfunktion som finns i vanliga databaser utan sökinformationen måste skapas och sparas undan i separat fält/tabell.

Bibliotek vill att om det är möjligt att presentera ett diakritiskt tecken, på skärm eller skrivare, så skall det göras. Ett bekymmer är att tangentbord har mer begränsade möjligheter att presentera stora teckensätt än vad en skrivare kan. Därför togs denna lösning fram, så att det går att registrera, visualisera och söka på tecken som inte finns representerade på ett tangentbord. Ett annat bekymmer är sorteringsordningen, mig veterligt finns det inget stöd i vare sig Unicode eller i de kommersiella databaserna att de stöder olika sorteringsalgoritmer beroende på vilket land som använder det, ett stort problem för de system som är globala och flerspråkiga.

Ett specialtecken lagras på följande vis

- **Prefix** – ett dollartecken som prefix
- **Grundbokstav** – Ta bort prickarna och Du får fram grundbokstaven, É -> E
- **Kod** – En enställig kod för diakriten

Exempel på koder är

- **Kolon :** - (Trema), ex Ö -> $O:, ö -> $o:
- **O –** (Ring), ex Å -> $AO
- **G** – Grekiska, ex ? (delta) -> $dG. Det finns en translittereringsstandard mellan grekiska och latin, så den används för grundbokstäverna

Denna lösning ger flera fördelar:

- **Enklare inmatning** – En katalogisör behöver inte lära sig tusentals koder för alla tänkbara diakritiska tecken. Skall ett Ë matas in och det inte finns på tangentbordet, letar man rätt på koden för det diakritiska tecknet (finns ca 10 stycken) och matar därefter in $E:
- **Enklare sökning** – Be någon söka efter Citroën. Är Du osäker på de diakritiska tecknen, skriv istället grundbokstäverna istället, i detta fall Citroen. Att detta fungerar beror på att (med några få undantag) indexet är lagrat utan diakriten utan endast med grundbokstaven. Innan sökningen startas, så tas alla diakriter bort från söksträngen innan databasen anropas
- **Enklare sortering** – Eftersom diakriterna hanteras som de gör och att indexen är skapade efter en filtrering, så kan man hantera ex danskans Å och AA som likvärdiga.

E: Felhantering

Felhanteringen skall leverera information till flera parter,

- **Anropande** – den som anropar en tjänst vill inte veta av exakt vad som är fel, utan mer; beror felet på mig eller Dig ? om jag gjorde fel, vad kan jag göra åt det ? om Du gjorde fel, är det allvarligt, kan jag sända om ?
- **Systemadministratören** – Han behöver dels den korrekta interna felkoden, ex från databasmotorn, dels den informationsmängd som funktionen hanterade när felet uppstod för att kunna återskapa felet
- **Helpdesk** – helpdesk behöver bra information för att snabbt kunna åtgärda allvarliga fel
- **Förvaltaren** – Han behöver bra information kring felen för att kunna ta ställning till om justeringar behöver göras i systemet

Felhanteringen sköts om av en generell klass/funktion. Den ser till att den returnerande Containern innehåller bra information till det anropande systemet och det loggas bra information tillsammans med den Container som systemet hanterade i den stund felet uppstod för att stödja de övriga på bästa sätt.

Ett problem som varje utvecklare står inför, det är när han skall ta hand om ett fel. Hur skall han handhava och vad för relevant information skall han sända till helpdesk, systemadministratören och slutanvändaren. Den problematiken sköter denna felhantering åt honom, genom att felhanteringsklassen tillsammans med en Container levererar intressant information till alla dessa parter.

Här följer en dokumentation över de felkoder som Schenker använder sig av i svaret från ett anrop till en Schenker tjänst. Att notera är att eftersom en Container kan hantera flera fält, så består Schenkers returinformation av ett antal fält, ex en vektor med inkomna variabler som hade något fel som felorsak för var och en av dem.

Obligatorisk fält är endast system.error.id, som har värdet noll om bearbetningen gick bra.

SCHENKERS FELKODER VID SVAR FRÅN EN TJÄNST

A lot of things can go wrong in an application. When it happens, the application sends error information to the system administrator and helpdesk. This information is usually not of interest for the calling client. The client only wants to know if the call went OK, or if there were any problems. If there were, was it because of the call or something inside the service. That's why the returning information is pre-defined and does not have so many different error codes.

The service could send a more descriptive text about an error, but this text is only for debug and logging purposes. It mustn't be shown to the end user, nor be interpreted programmatically by the client, since it may change at any time.

A service can have its own error codes (unique for the service and should be an exception). These must be a more detailed description of one of the pre-defined error codes and shall be set in system.error.service.error_id, instead of system.error.id where the pre-defined is set.

Sometimes, the call went OK (system.error.id=0) but the service want to send some warning to the caller. This warning should be part of the data_list but it could be send in system.error.service.error_id.

A service can generate an unique key when an error occurs (for every call), which it sends to helpdesk and system administrator. This key can also be sent back to the caller, so it is possible to have the same error identification on both sides.

Add 'system.error.' to all fields, e.g. id -> system.error.id

Name	O	Type	Explanation
id	M	Integer	Error code. All legal values defined below.
message		String	Error message. Standard message for 'id'. Only used for debug or system logs. If used, it should be the same as defined below for 'id' explanation.
description		String	A more detailed description of what is wrong. Only used for debug or system logs.
key		String	Unique key from the service about this error. It helps system administrators to find the same error message in different logs.
service.error.id		Integer	Unique service error code. If the service needs a more detailed error code than the defined one, the service can define their own. Must be used as a more detailed predefined error code. It could also include a warning code when system.error.id=0.
service.error.message		String	Service error message. Standard message for 'service.error_id'. Only used for debug or system logs..
service.error.description		String	A more detailed description of what is wrong. Only used for debug or system logs.
validation_list		Array	Information about the fields, which have validation errors. Only used with error code 210.

Validation error

The fields that have some sort of validation errors are returned in this array. Only used when system.error.id=210.

Add 'system.error.validation_list.n.' to all fields.
E.g. field_name -> system.error.validation_list.n.field_name (where n >= 0)

Name	M	Type	Explanation
field_name	M	String	The name of the field, in the HQF (HTTP Querystring Format) including the whole path. E.g request.service.action
error.id	M	Integer	Error code in 300 series
error.message		String	Error message. Standard message for 'id'. Only used for debug or system logs. If used, should be the same as defined below for 'id' explanation.
service.error.id		Integer	Unique service error code. If the service needs a more detailed error code than the defined one, the service can define their own. Must be used as a more detailed predefined error code.
service.error.message		String	Service error message. Standard message for 'service.error.id'. Only used for debug or system logs.
old_value	M	All	The old wrong value
from_value		All	Lowest value the field could have.
to_value		All	Highest value the field could have.

For error code 311 (use only one suggestion), you can use 'field_list' to show which fields you have to choose between. You have to repeat the field named in 'field_name' above to the 'field_list.0.field_name'.

E.g. the user could choose between a region and a city. If the user chooses both and that generate an error, you first define the error for one of the fields, say the region and then you

send all fields that couldn't be used together, in this case the region and the city. The error message could be:

system.error.id=210 (validation error)
system.error.validation_list.0.error.id=311 (only one suggestion)
system.error.validation_list.0.field_name=request.select.region
system.error.validation_list.0.field_list.0.field_name=request.select.region
system.error.validation_list.0.field_list.1.field_name=request.select.city

Add 'system.error.validation_list.n.' to all fields.

Name	M	Type	Explanation
field_list		Array	Only used with error code 311 to show which fields the user has to choose between.
field_list.n.field_name		String	The name of the field, in the HQF (HTTP Querystring Format) including the whole path.

Service closed

If a service is closed because of some planned action

Add 'system.error.' to all fields.

Name	M	Type	Explanation
id	M	Integer	Error code 10
message		String	"Service closed"
closed.from_date_time	M	DateTime	DateTime when the service was closed.
closed.to_date_time	M	DateTime	DateTime when the service is planed to be open again.
closed.message		String	A message in English that describes that the service is closed and when it is planed to be open again.

Error codes

These codes are used in 'service.error.id' except for the 300 series, which are used in 'system.error.validation_list' when error code 210 is used. These are the only legal choices. If a service need to use one service defined error code, it must be used in the field 'system.error.service.error_id' or 'system.error.validation_list.n.service.error_id'.

If the service want to send some warning information, this is also done in 'system.error.service.error_id' while returning system.error.id=0.

OK

Code	Meaning	Explanation
0	OK	All went well

Service closed

Code	Meaning	Explanation
10	Service closed	The service is closed for some planned action.

Authorization

Code	Meaning	Explanation
101	Permission denied	Are not allowed to do the action.
102	Wrong login and password	Can't identify the user. Could also be wrong certificate.
103	ID locked by other user	Other user locks the required information.

Incorrect call

Code	Meaning	Explanation
201	Can not find service	The service doesn't exist
204	Invalid container	The container has a bad format or is too large.
205	Could not perform action	The service could not do its job. The difference between this error and the 500-series is that 500-series is about fatal errors and should invoke helpdesk or a system administrator to correct something and the 205-error should not involve them. E.g. Send a 205-error when there is an insert action but the id are already occupied.
210	Invalid data	Incoming field (or several) has a wrong value. A more detailed description can be done in 'system.error.validation_list' and with error codes from the 300 series.

Validation error

These error codes are used in 'service.error.validation.n.id' together with error code 210.

Code	Meaning	Explanation
301	Invalid format	Bad format or illegal characters. E.g. bad date format or letters in a numeric field.
302	Illegal value	The value is out of range or not legal.
304	Mandatory field is missing	The mandatory field is missing or have no value.
309	Illegal index	The array index is illegal
310	Illegal length	The length of the value is too long or to short. You can send the max length in to_value'. Normally when a value is to long, it should be truncated instead.
311	Only one suggestion	Used when the user have to choose between one of several fields. The fields could be named in 'field_list.n.field_name'

Fatal error

These errors means that the service can't do its job and – probably – a system administrator has to do some work to make the service up and running again. The service shall send a message to helpdesk when this occurs.

Code	Meaning	Explanation
501	Connection timed out	The request took too long to execute and was timed out.
506	Backend system unavailable	Central system, the database, etc, doesn't response.
507	Internal server error	All other fatal errors
508	Internal data error	Data error in the backend system