
# Serviam: Proof of Concept-rapport

2005-11-04

*Editor*: Eva Söderström

*Bidrag av*: Milena Haykowska, Jesper Holgersson,
Rahel Hussain, Jana Kjebon, Magnus Larsson,
Oddgeir Vestad, samt Jelena Zdrakovic

SERVIAM-POC-01

Version 1.1

# Innehållsförteckning


**1 INTRODUKTION 1**

1.1 MÅL OCH SYFTE MED RAPPORTEN 1

1.2 POC – PRESENTATION AV GRUNDIDÉN 1

1.3 SAMMANFATTNING AV RESULTATEN 1

*1.3.1. Slutsatser 1*

*1.3.2. Rekommendationer 2*

1.4 STRUKTUR PÅ RAPPORTEN 2

**2 AKTIVITETER 3**

2.1 ÖVERGRIPANDE AKTIVITETSMODELL FÖR ARBETET MED POC 3

2.2 DELAKTIVITETER 3

*2.2.1. Processmodell 3*

*2.2.2. Säkerhetsrekommendationer 4*

*2.2.3. Web Service-design 4*

*2.2.4. ITeas Web Service 4*

*2.2.5. SEBs Web Service 4*

*2.2.6. Realisering och testning 4*

*2.2.7. Avrapportering 4*

**3 RESULTAT 5**

3.1 ÖVERSIKT ÖVER DEN GEMENSAMMA WEB SERVICEN 5

*3.1.1. Kort process beskrivning 5*

*3.1.2. Webbtjänstens operationer 6*

*3.1.3. Gränssnittsdesign av webbtjänsten 6*

*3.1.4. Säkerhetsrekommendationer 7*

3.2 SEBS PERSPEKTIV 8

*3.2.1. Processmodell 9*

*3.2.2. Teknisk lösning 10*

3.3 ITEAS PERSPEKTIV 10

*3.3.1. Processmodell 10*

*3.3.2. Teknisk lösning 11*

**4 PROBLEM 15**

4.1 SÄKERHET 15

*4.1.1. Tidsaspekten 15*

*4.1.2. WSS kontra SEBs utvecklingsmiljö 15*

*4.1.3. WSS kontra ITeas utvecklingsmiljö 15*

*4.1.4. Rekommendationer gällande säkerhet 16*

4.2 BEGREPP 16

*4.2.1. Problem 16*

*4.2.2. Rekommendationer gällande begrepp 16*

4.3 IMPLEMENTATIONSSYSTEMEN 16
*4.3.1. Tidsaspekten* *17*
*4.3.2. Systemkomplexitet och kompatibilitetsproblem* *17*
*4.3.3. Rekommendationer gällande implementationssystem* *17*
4.4 PROCESSEN 18
*4.4.1. Problemet* *18*
*4.4.2. Rekommendationer gällande processer* *18*
**5 RELATION TILL MÖNSTERKATALOGEN 19**
5.1 RELATION TILL MÖNSTERKATALOGEN 19
*5.1.1. Juridikmönster* *19*
*5.1.2. Planeringsmönster* *20*
*5.1.3. Upptäcktsmönster* *20*
*5.1.4. Kompositionsmönster* *21*
*5.1.5. Säkerhetsmönster* *21*
*5.1.6. Kommunikationsmönster* *23*
**6 SLUTSATSER 24**
6.1 SÄKERHET:SMÄSSIGT 24
6.2 TEKNISKT 24
6.3 UTVECKLINGSPROCESSEN 24

## BILAGOR

Bilaga 1: Detaljerad specifikation av webbtjänstens operationer
Bilaga 2: Gränssnittsdesign
Bilaga 3: Säkerhetsrekommendationer
Bilaga 4: Exempel på BPEL-kod för WEB SERVICE
Bilaga 5: Kodexempel för säkerhet
Bilaga 6: Javakod för Web Service-gränssnittet
Bilaga 7: Utveckling av klienten i ITea-SEB projektet
Bilaga 8: Jar-filer som måste anges i class path

# 1 Introduktion

Introduktionskapitlet visar på mål och syfte med rapporten, en presentation av grundidén till proof of concept (PoC), samt en sammanfattning av resultaten.

## 1.1 Mål och syfte med rapporten

Rapporten syftar till att presentera ett genomfört ”Proof of Concept” (PoC) inom ramen för projektet Serviam. Innehållet i PoC presenteras i avsnitt 1.2. Rapporten fokuserar på genomförda aktiviteter, uppnådda resultat och påstötta problem. Delvis kommer även resursåtgång kort att omfattas.

## 1.2 PoC – presentation av grundidén

I projektet Serviam har konceptet tjänsteorienterade arkitekturer utforskats och belysts från olika perspektiv: säkerhet, arkitektur, affärsnytta och förvaltning. I litteraturen finns många löften om vad SOA och framför allt webbtjänster innebär och bringar. Däremot finns få empiriska bevis på att detta fungerar. Inom ramen för Serviams andra år utvecklades därför ett *proof of concept*, för att visa på hur det kan fungera att få till en webbtjänst mellan två organisationer med vilka problem etc som kan uppstå under processen. De två inblandade ”organisationerna” är SEB och Stockholms Universitet/KTH. I den sistnämnda var ”organisationen” ett fiktivt möbelföretag med möjlighet att beställa via webben.

## 1.3 Sammanfattning av resultaten

Resultaten från *proof of concept* kan beskrivas både i termer av slutsatser och i termer av rekommendationer. Båda kommer här kort att beröras.

### 1.3.1. Slutsatser

Slutsatserna kan delas in i tre delar: säkerhet, teknik och utvecklingsprocess.

| | |
|---|---|
| Säkerhet | Även om dagens utvecklingsmiljöer ofta hävdar stöd för standarden WSS är inte detta alltid fallet. Ett exempel är ”stöd” för aspekter i WSS roadmap som ännu inte är fullt standardiserade. Konsekvensen kan bli olika implementationer och tolkningar av WSS i olika miljöer, och standarden är då inte längre en standard. Kompatibiliteten kan då ifrågasättas. |
| Teknik | Enbart WSDL fungerar rent tekniskt. Komplexiteten kommer in på två nivåer, dels tekniskt med UDDI, och dels affärsmässigt med avtalsbiten. Vision och verklighet matchar inte ännu till 100%, eftersom det inte är så lätt som det utger sig från att vara. |
| Utvecklingsprocess | Olika valsituationer behöver hanteras under utveckling och design av Web Services. Exempel är processgranularitet; specificering av processimplementationer; top-down eller bottom-up-ansats; fel- och transaktionshanteringsprocedurer att definiera; m.m. Utvecklingsprocessen kan komma att innebära nya typer av problem. |

### 1.3.2. Rekommendationer

Rekommendationerna gäller fyra aspekter: säkerhet, begrepp, implementationssystem, samt processer.

| | |
|---|---|
| Säkerhet | Undersök hur befintliga arkitekturer, principer och beslut gällande säkerhet påverkar möjligheten att använda och implementation av t.ex. WS Security. Utveckling av en bra säkerhetslösning kräver tid. |
| Begrepp | För att Web Services ska fungera i den befintliga miljön bör WSDL-filen för alla Web Services som utvecklats av andra parter studeras för att utröna hur centrala begrepp definieras. För att säkerställa en gemensam begreppsapparat i B2B-sammanhang bör en gemensam begreppsanalys genomföras för att undvika tvetydigheter. |
| Implementation | Valet av SOAP-implementation är inte trivialt och analys av alternativ bör göras grundligt för att undvika problem i senare skeden. Web Services-tekniken lider fortfarande av barnsjukdomar och kräver sålunda en viss ansträngning för att fungera. Arkitekturer, policybeslut, med mera, behöver ses över för att avgöra hur de påverkar skapandet och användningen av Web Services. |
| Processer | Verktyg för exekverbara processer i BPEL4WS brottas fortfarande med tekniska bekymmer, vilket kan försvåra och försena övergången till sådana verktyg. |

## 1.4 Struktur på rapporten

Rapporten är uppdelad i tre större delar: aktiviteter (kapitel 2), resultat (kapitel 3) och problem (kapitel 4). Den första delen fokuserar på hur PoC-arbetet bedrevs. Den andra visar på vad aktiviteterna resulterade i. Kapitel 4 visar på vilka problem som uppstod på vägen och hur dessa hanterades. Efter de tre huvuddelarna följer ett summeringskapitel (kapitel 5) och ett slutsatskapitel (kapitel 6).

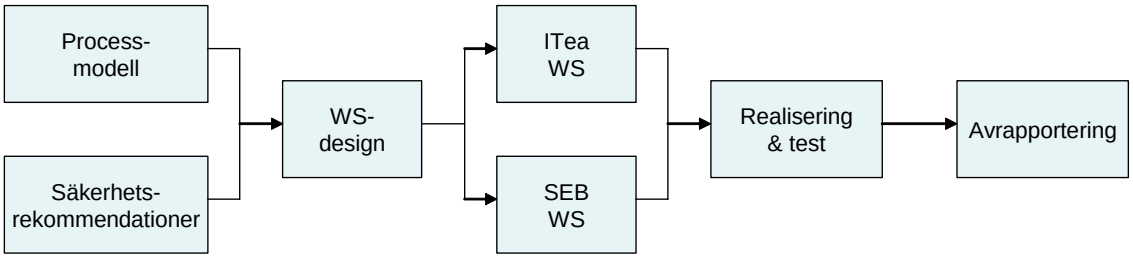
Arbetet rapporteras med utgångspunkt från den kronologiska aspekten, där aktiviteterna är i centrum, medan aktörer och resurser relateras till aktiviteterna. Det huvudsakliga resultatet är en fungerande Web Service. Denna beskrivs först övergripande, och sedan utifrån respektive deltagande organisations perspektiv. De problem som uppkom har kategoriserats utefter deras natur, för att sedan diskuteras och utvecklas.

# 2 Aktiviteter

Här ska PoC olika delsteg presenteras. Någon form av aktivitetsmodell/processmodell ska presenteras över hur vi har gått tillväga. Detta ska inkludera vilka aktörer som har gjort vilka delar av processen/processerna. Respektive delprocess i modellen ska också beskrivas, möjligen i varsitt underkapitel. Delavsnitt presenteras nedan.

## 2.1 Övergripande aktivitetsmodell för arbetet med PoC

Arbetet med PoC har genomgått sju steg (se figur 1). Till att börja med definierades flödet mellan de två organisationerna, vilket dokumenterades i processmodeller. Samtidigt togs två förslag fram till vilken säkerhetslösning som skulle kunna vara aktuell för lösningen. Dessa två steg skedde alltså parallellt. Därefter gjordes en första design av webbtjänsten, med definition av dess operationer, samt konstruktion av ett klassdiagram.




**Figur 1: Processmodell över Serviams PoC**

Från den initiala designen gjorde de två organisationerna arbeten internt med de egna systemen för att realisera webbtjänsten. Även här var det alltså två parallella steg. Efter att de tekniska lösningarna hade bearbetats i respektive organisation gjordes sammankopplingarna och Web Servicen kördes live. I detta ingick att testa huruvida kopplingen fungerade, med påföljande felletande i kod, etc. Slutligen gjordes en avrapportering där denna större rapport, samt en mindre skrift riktad utanför projektet framställdes. Respektive delprocess kommer att definieras och beskrivas i mer detalj i avsnitt 2.2. Utgångspunkten kommer att vara de olika delaktiviteter som tillsammans har fullgjort varje del. I slutet av rapporten (avsnitt 5.1) sammanfattas hela projektet i en stor tabell (tabell 1, avsnitt 5.1). Denna fokuserar på varje deltagares aktiviteter, medan 2.2 istället fokuserar på aktiviteterna som sådana.

## 2.2 Delaktiviteter

Samtliga sju steg i vårt PoC-arbete kommer i detta delavsnitt att beskrivas med utgångspunkt från primära aktiviteter.

### 2.2.1. Processmodell

Mycket av arbetet skedde i form av diskussioner, där frågorna rörde sig kring dels vilka krav ITea ställde på SEBs Web Service (in- och utparametrar, vad tjänsten skulle göra), samt

dels hur länkar mellan parterna skulle specificeras. Respektive aktör gjorde en modell över sin del av samverkan, enligt överenskomna variabler och processlogik, samt med överenskommet BPEL-verktyg.

### 2.2.2. Säkerhetsrekommendationer

För att kunna bestämma lämplig säkerhetsnivå studerades först möjligheterna i standarden Web Services Security, följt av framtagning av två alternativa säkerhetslösningar. I detta fördes diskussioner kring vad i standarden som redan stöttas hos SEB med tanke på befintlig teknik och lösningar.

### 2.2.3. Web Service-design

Baserat på den utförda processmodelleringen skapades ett gränssnitt för Web Servicen, där operationerna specificerades. Därefter designades implementationsklasser för SEBs tjänster, innan de implementerades genom kodning i Java.

### 2.2.4. ITeas Web Service

Från ITeas sida utvecklades och testades klienten enligt specifikationerna, först utan Web Services Security. Därefter gjordes en utökning till att även innefatta standarden.

### 2.2.5. SEBs Web Service

Inom SEB togs ett kravdokument fram. Olika SOAP-implementationer studerades innan Axis valdes framför WebSphere. Till Axis valdes WSS4J. Efter detta uppkom problem med SOAP-implementationen, och beslut fattades att byta till WebSphere. Den grundläggande Web Servicen utökades med ett gränssnitt, vilket inkluderade ett deploymentdiagram och en plattformsbeskrivning. Från Web Services Security lades *user name password token profile* på. Samtidigt med detta utvecklades också en intern miljö för att deploya Web Servicen på inom SEB.

### 2.2.6. Realisering och testning

Under realiseringen upptäcktes ett fel i logiken. Detta föranledde felsökning i koden. Efter att felet korrigerats gjordes en ny provkörning/deployment.

### 2.2.7. Avrapportering

Information samlades in från samtliga aktörer i projektet, både genom möten och genom insamling av dokumentation av arbetet enligt en förberedd mall. Materialet sammanställdes och två olika rapporter färdigställdes (varav denna är en).

# 3 Resultat

Här presenteras lösningen (Web Servicen mellan ITea och SEB) kort. Det börjar med en gemensam översikt, innan mer detaljer läggs till från respektive perspektiv.

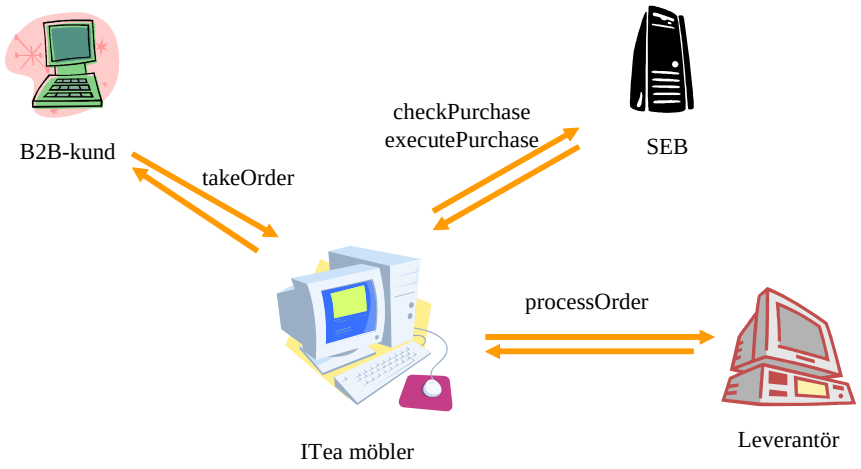
## 3.1 Översikt över den gemensamma Web Servicen

Den webbtjänst som har konstruerats måste distribueras med klientcertifikat för användning.

### 3.1.1. Kort process beskrivning

Exemplet gäller beställning av möbler från webbutiken ITea (http://ITea.dsv.su.se, se en översikt i figur 2). Kunden kan placera en beställning genom att fylla ett köpformulär som ligger på Iteas Web plats. I formuläret fyller kunden i följande information:

- Specifikation av möbler som tas från en lista.
- Betalningsinformation, dvs. kreditkortsdetaljer.
- Information om leveransadressen.




**Figur 2: Översikt av ITeas affärer.**

Information från formuläret tas emot i form av en Web service (WS_I). ITea behandlar kundens information genom att:

- Beräkna beställningssumman, internt.
- Kontrollera betalningsmöjligheter, genom att kontakta kundens banks (SEB) Web Service (WS_SEB). Om summan godkänns av SEB fortsätter processen, annars får kunden ett negativt svar med en detaljerad förklaring av problemet. I det fallet kan kunden ändra information i formuläret och begära köpet igen.
- Om summan godkänns av banken kontaktar ITea leverantören genom Web Services (WS_L) och kontrollerar om varorna finns och kan levereras. Om leverantören

godkänner beställningen fortsätter processen, annars ges kunden ett negativt svar och processen avslutas.

- Om leveransen är godkänd kontaktar ITea banken igen (WS_SEB) med kravet att dra summan från kundens konto. Om tjänsten är godkänd får kunden ett positivt svar, dvs. beställningen bekräftas. Om inte annulleras beställningen till leverantören, kunden får att negativt svar och processen avslutas.

### 3.1.2. Webbtjänstens operationer

Den framtagna webbtjänsten innehåller följande operationer:

WS_I: operation **takeOrder** – tar information från kunden om köpet.

WS_SEB: operationer checkPurchase, executePurchase

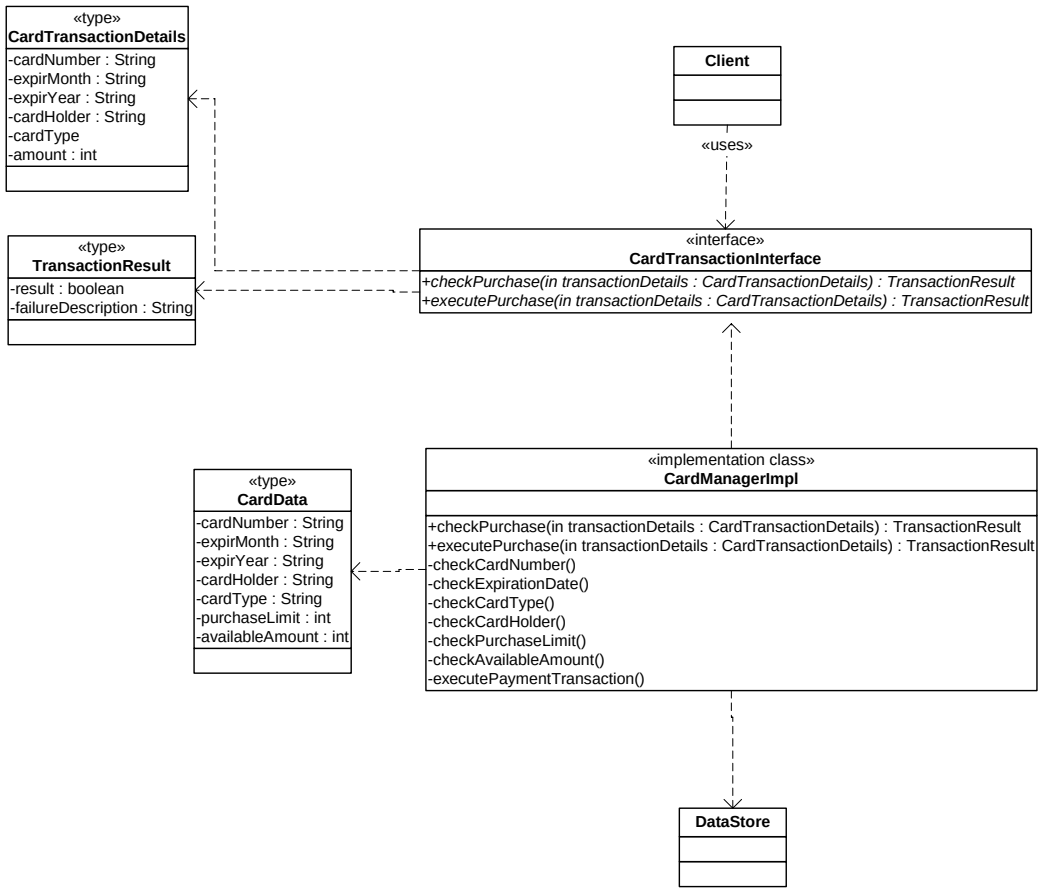
- **checkPurchase**: Kontrollerar att kortet som kunden vill betala med är giltigt och att beloppet för detta köp finns tillgängligt
- **executePurchase**: Utför samma kontroller som i checkPurchase, om allt är OK utförs en betaltransaktion.

WS_L: operation **processOrder** – Beställer varor från leverantören.

Operationerna beskrivs mer i detalj i Bilaga 1. Processmodellerna för webbtjänstens användning och det som sker i de båda organisationerna är starkt relaterat till operationerna. Processmodellen för SEB beskrivs i avsnitt 3.2.1 och processmodellen för ITea beskrivs i avsnitt 3.3.1. Båda modellerna beskrivs med notationen Business Process Modelling Notation (BPMN).

### 3.1.3. Gränssnittsdesign av webbtjänsten

Teknisk gränssnittsdesign, databärande klasser samt implementationsklasser för operationer checkPurchase och executePurchase visas i figur 3.




**Figur 3: Teknisk gränssnittsdesign för PoC**

I och med att figuren är relativt liten finns den i en något förstorad form i Bilaga 2. Där finns också en mer detaljerad bild med gränssnittsdesign, databärande klasser samt implementationsklasser för samtliga operationer.

### 3.1.4. Säkerhetsrekommendationer

Ett led i PoC var en diskussion av säkerheten i webbtjänsten. Syftet var att ge förslag på hur säkerheten skall hanteras när överföringar mellan ITEA och SEB görs. Önskemålet var en så enkel lösning som möjligt, där SEB efterfrågade begränsad användning av certifikat.

Resultatet från arbetet var att luta sig på förslag 1 (se bilaga 3) och innebär kortfattat att:

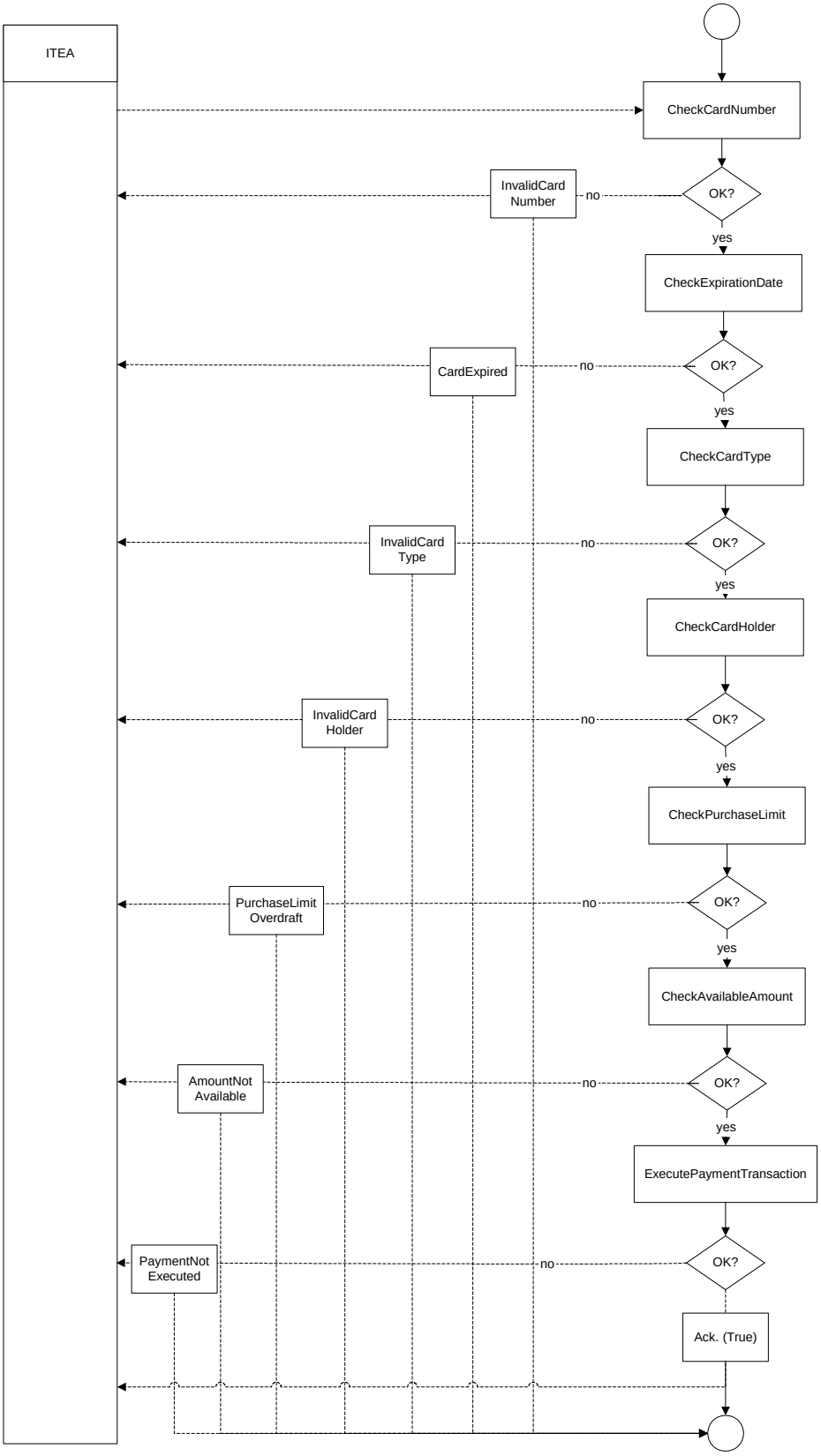
1. Autentisering sker med ett, för båda sidor, känt lösenord som kombineras med ett slumpvärde (digest) och en tidsstämpel och tillsammans bildar en *hashfunktion*.
2. Kontokortsnumret som skall överföras mellan aktörerna krypteras med XML-encryption på symmetrisk basis.
3. All kommunikation kommer att ske i ett skal av SSL. (Detta ingår ej i lösningsförslaget men är ett krav som framkommit senare från SEB. Detta lager kommer dock att vara transparant för betraktaren.)

Lösningen beskrivs i mer detalj i bilaga 3.

## 3.2 SEBs perspektiv

Detta avsnitt presenterar två aspekter utifrån SEBs perspektiv: en processmodell över SEBs interna del av samarbetet (3.2.1), samt en kortare beskrivning av den tekniska lösningen (3.2.2). Relaterat till processmodellen finns också en del BPEL-kod för orkestreringslogik. Ett exempel på denna finns i Bilaga 4.

### 3.2.1. Processmodell




**Figur 4: SEBs interna process.**

### 3.2.2. Teknisk lösning

SOAP-implementationen i grunden är WebSphere application server 6.0. Applikationsservern som detta körs på är även det WebSphere. Tillägget WSS4J (Web Services Security for Java) användes för att uppnå säkerhet. Denna bygger på olika open source-produkter, som till exempel en XML-transformer.

User name password token profile, den enklaste standarden i WS Security (WSS), lades på. Det hade varit att föredra att kunna lägga på ytterligare säkerhet i form av ett klientcertifikat (x.509 token profile), men detta krävde för mycket tid. I Bilaga 5 finns ett kodexempel på hur säkerhet kan hanteras. Där beskrivs dels en kort fil om hur önskad säkerhetskonfiguration kan anges, och dels en kort fil hur densamma implementeras.

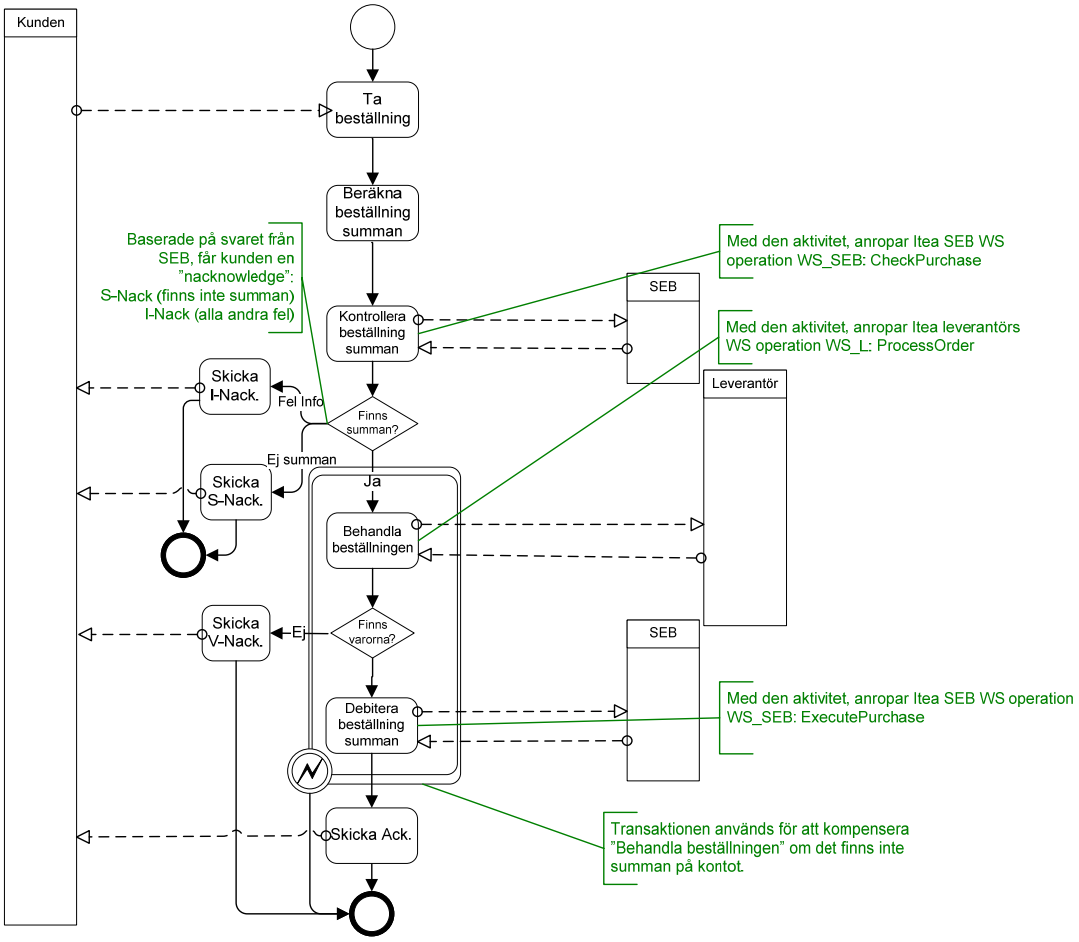
I Bilaga 6 finns även en bit kod som beskriver Javagränssnittet (user interface) för den framtagna Web Servicen.

## 3.3 ITeas perspektiv

Detta avsnitt presenterar två aspekter utifrån ITeas perspektiv: en processmodell över ITeas interna del av samarbetet (3.3.1), samt en kortare beskrivning av den tekniska lösningen (3.3.2).

### 3.3.1. Processmodell

I figur 5 visas processen från ITeas perspektiv.




**Figur 5: ITEAs affärsprocess.**

### 3.3.2. Teknisk lösning

En Java-klient har skapats i J2EE 1.4-plattformen, med hjälp av Java Web Services Developer Pack 1.5 (JWSDP). Klienten anropar SEBs webbtjänst med WS-Security User name password token.

Klienten har sedan integrerats med ITea systemet och installerats på en JBoss applikationsserver. När en kund lägger en beställning via ITea webbsajten anropas SEB tjänsten för att kontrollera kundens kontokortsinformation. Om informationen godkänns av SEB godkänns också beställningen. För mer detaljer, t.ex. i form av kod, se Bilaga 7 (Utveckling av klienten i ITea-SEB projektet).

# 4 Problem

Detta kapitel innehåller en beskrivning av olika problem som uppkom under arbetets gång. Dessa problem har delats upp i fyra delar: säkerhetsproblem (avsnitt 4.1), begreppsproblem (avsnitt 4.2), implementation och system (avsnitt 4.3), samt processrelaterade problem (avsnitt 4.4).

## 4.1 Säkerhet

Säkerhetsproblem relaterar antingen till standarden (Web Services Security – WSS) i sig eller till arbetet med att implementera densamma. Det betyder att det antingen var tidsfaktorn som spelade in, eller olikheter i WSS-specifikationerna kontra SEBs respektive ITeas utvecklingssmiljöer.

Värt att nämna är att inga problem upplevdes som större, utan det som framkommit var förväntade problem. En bra dialog mellan inblandade parter har medfört att de lösningar som tagits fram är till belåtenhet för alla.

### 4.1.1. Tidsaspekten

På grund av att det fanns begränsat med tid för att genomföra PoC kunde inte säkerhetsfunktionen bli på den nivå som egentligen önskades. Nuvarande lösning är den enklast möjliga. Det hade dock varit önskvärt att ytterligare nivåer av säkerhet hade lagts på. Dessa skulle ha varit:

- Autentisering: med X.509 token profile, för påloggning
- Kryptering (WS Encryption), för insynsskydd
- Signering (WS Signature), för att meddelanden inte ska kunna förändras

Hade det varit ett riktigt projekt så skulle mer tid ha lagts ned. Ändå tog PoC mer tid än förväntat.

### 4.1.2. WSS kontra SEBs utvecklingsmiljö

Alla förslag från Web Services Security (WSS) kunde inte implementeras rakt av på grund av olikheter i specifikationen för WSS kontra stödet i SEBs utvecklingsmiljö. Specifikationen för WSS säger en sak, medan stödet i SEB:s utvecklingsmiljö säger en annan. Det går alltså inte bara att ge förslag direkt från WSS och tro att detta skall kunna genomföras rakt av.

Dessutom begränsas användningen av WSS av att certifikatanvändning inte är önskvärt från SEBs sida. Miljön som allt ska implementeras i är rätt speciell, vilket medför ett behov av flera ”ad hoc”-anpassningar. Ett sådant exempel är SSL.

### 4.1.3. WSS kontra ITeas utvecklingsmiljö

Det primära problemet med ITeas utvecklingsmiljö och WSS var att verktyget ”WS Compile”, som genererar klientproxyn, inte ville ta hänsyn till säkerhetskonfigurations-filen. Denna fil innehåller information om vilken typ av säkerhetsteknik som ska användas, inklusive användarnamn och lösenord. Problemet var att WS Compile inte godkände att

security-flaggan angavs då verktyget exekverades, detta trots att alla nödvändiga jar-filer angavs i classpath. En möjlig orsak till problemet kan vara att det kan ha funnits några äldre varianter av de nödvändiga jar-filerna i några andra kataloger som ingick i miljövariablen classpath. Ett försök till att skapa en Apache AXIS-klient gjordes, med WSS4J. Det var krångligt att få detta att fungera, trots att anvisningar och manualer följdes. Två olika datorer testades utan att orsaken hittades, innan ett tredje försök på en tredje maskin lyckades. När det gäller jar-filerna finns ytterligare problem och information i avsnitt 4.3.2.

### 4.1.4. Rekommendationer gällande säkerhet

Undersök hur befintliga arkitekturer, principer och beslut gällande säkerhet påverkar möjligheten att använda och implementation av t.ex. WS Security. Utveckling av en bra säkerhetslösning kräver tid.

## 4.2 Begrepp

Avsnittet innehåller dels en beskrivning av problemet som uppkom, och dels en rekommendation utifrån upplevd problematik.

### 4.2.1. Problem

När företag ska samarbeta krävs att de är överens om innebörden i en del centrala begrepp. Om inte detta görs kan både personal och system missförstå varandra och fel uppstå. I PoC har inte begreppsförvirring varit ett stort problem. Det enda begrepp som gav upphov till viss förvirring mellan SEBs och ITeas system var ”Kort-ID”, framför allt gällande vad som skulle ingå i det (enligt SEBs krav). Det blev mycket trial-and-error, trots att tanken med denna typ av arkitektur är att det ska gå enkelt att göra utan beskrivningar. Sådant behövs dock i praktiken, t.ex. för att inparametrar inte syns i WSDL-filer. Kontentan är att det krävs en gemensam terminologi när man arbetar tillsammans för att inter-organisatoriskt samarbete ska fungera.

### 4.2.2. Rekommendationer gällande begrepp

Om en organisation ska implementera Web Services som har utvecklats och därmed ägs av ett annat företag, så behöver WSDL-filen studeras för att utröna hur centrala begrepp definieras. Bland annat är detta nödvändigt för att Web Services ska fungera tillsammans med den befintliga miljön.

I ett Business-to-Business (B2B) sammanhang behöver partners som ska samarbeta eller nyttja varandras webbtjänster genomföra en begreppsanalys för att säkerställa att en gemensam begreppsapparat används och att tvetydigheter undviks.

## 4.3 Implementationssystemen

Detta var den största kategorin av problem och den kan delas in i två delar: tidsaspekten, samt systemkomplexitet och kompatibilitetsproblem.

### 4.3.1. Tidsaspekten

En grund för PoC Web Servicen var att ha en SOAP-implementation. Till en början beslutades att open source-produkten Apaches AXIS skulle användas. Dock fungerade det inte optimalt, dels med WebSphere som applikationsserver (vilket är den server som SEB använder), men även dels med säkerhetstillägget WSS4J. Samtidigt kom också en så kallad ”red book” för WebSpheres egen SOAP-implementation, och ett val gjordes att byta till WebSphere från AXIS. Detta tog tid, särskilt när mer säkerhetsfunktioner skulle läggas till.

### 4.3.2. Systemkomplexitet och kompatibilitetsproblem

Som angavs i 4.3.1 var det ursprungliga valet av SOAP-implementation Apache AXIS. WebSphere var dock med i diskussionen från början, men bedömdes vara för komplext, odokumenterat och nytt vid tidpunkten för valet. Dessutom är GUI i WebSphere designat för återanvändning, vilket gör det betydligt mer komplext än vad AXIS var. För att använda WebSphere behöver ett antal konfigurationsfiler gås igenom, som i sin tur bygger på relativt komplexa XML-filer. Detta är relativt lätt att göra i AXIS, medan man i WebSphere måste gå runt det stora GUI.

Ett problem som uppstod rörde att få verktyget *wscompile* (som genererar klientproxyn) att ta hänsyn till säkerhetskonfigurationsfilen som talar om vilken typ av säkerhetsteknik som ska användas. Detta testades på två PC-maskiner, men verktyget ville inte godkänna att security-flaggan angavs vid exekveringen av verktyget trots att alla nödvändiga jar-filer angavs i classpath. Några försök gjordes att skapa en klient med Apache AXIS och WSS4J som tillhandahåller säkerhetsfunktioner för AXIS API. Det var dock något krångligt att få detta att funka, trots att anvisningar följdes och manualer gicks igenom i detalj. Lyckligtvis så visade det sig att *wscompile* med security-flaggan fungerade utan problem på själva deployment-maskinen. Vad felet berodde på är inte klarlagt. I Bilaga 8 beskrivs de jar-filer som måste anges i classpath vid kompilering och exekvering av klientapplikationer som använder sig av Java Web Services Developer Pack 1.5’s (JWSDP) API.

Miljön som allt ska implementeras i är rätt speciell, vilket gör att flera *ad-hoc* anpassningar måste göras, t ex SSL. Detta beror på SEBs policy att allt som ska innanför deras brandväggar ska köras via SSL. Därmed fick också den initiala lösningen köras på det viset, men bara som ett extra lager utan att vara med som en del av lösningen.

Ett tillägg till den ursprungliga SOAP-implementationen var WSS4J. Arbetet med det gick bra tills mer säkerhet skulle läggas på utöver basic authentication. WSS4J är open source och bygger på andra open source-produkter. Ett exempel är en XML-transformer som visade sig inte vara kompatibel med applikationsservern WebSphere, på vilken AXIS kördes. Detta bidrog till att AXIS kastades ut till förmån för SOAP-implementationen i WebSphere.

### 4.3.3. Rekommendationer gällande implementationssystem

Valet av SOAP-implementation är inte trivialt och analys av alternativ bör göras grundligt för att undvika problem i senare skeden.

Ett syfte med Web Services är att det ska vara t.ex. plattformsoberoende och vara enkelt att få till. Dock finns det fortfarande barnsjukdomar med tekniken och det kräver en viss ansträngning att få det att fungera.

Precis som med säkerhet behöver arkitekturer, policybeslut, med mera, ses över för att bestämma i vilken mån och på vilket sätt de påverkar skapandet och användningen av Web Services.

## 4.4 Processen

Avsnittet innehåller dels en beskrivning av problemet som uppkom, och dels en rekommendation utifrån upplevd problematik.

### 4.4.1. Problemet

De processrelaterade problemen hänrör sig till övergången från BPMN-beskrivningen till en exekverbar BPEL4WS-process. Det betyder att det handlar om processmodelleringsspråk snarare än om processerna själva. Detta inkluderar även många tekniska bekymmer, som integrering av Web Services som utvecklats på olika plattformar med verktyget för BPEL4WS som använts. Nuvarande resultat är att processen fungerar – om än i sin mest grundläggande form (utan felhanteringsprocedurer). Orsaken är att tiden inte räckt till för att lägga till ytterligare funktioner och procedurer.

### 4.4.2. Rekommendationer gällande processer

Organisationer som ska använda verktyg för att övergå från processer beskrivna i BPMN till exekverbara dito i BPEL4WS bör vara medvetna om att verktygen ännu brottas med tekniska bekymmer, särskilt om Web Services ska integreras som utvecklats på olika plattformar.

# 5 Relation till mönsterkatalogen

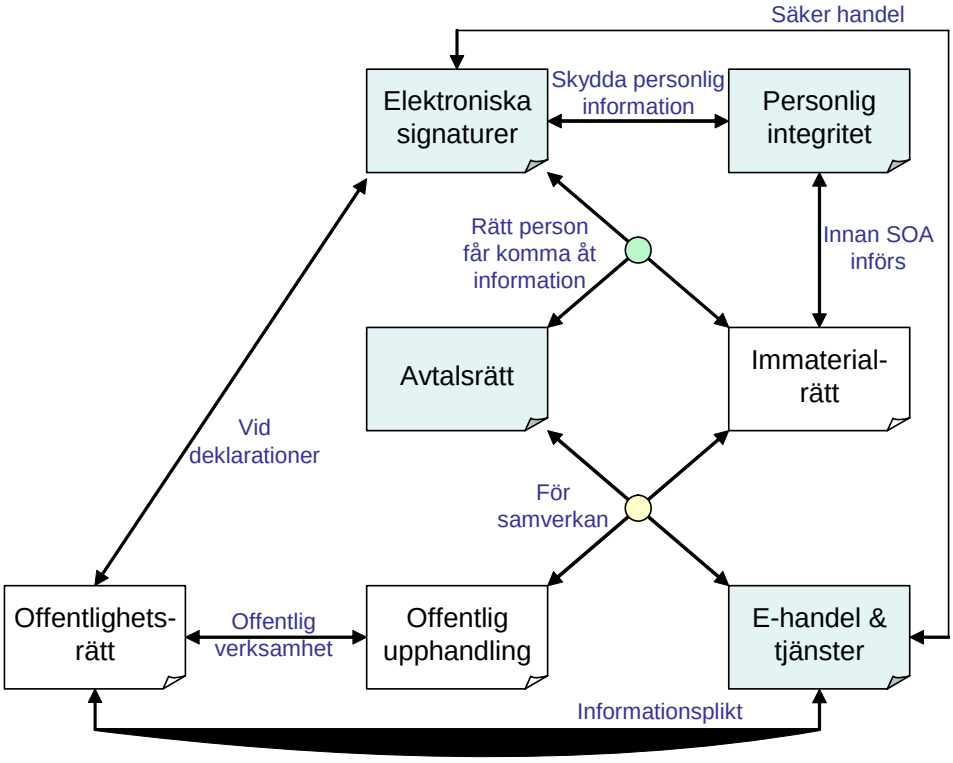
Kapitlet fokuserar på att kort relatera framkommet material till den mönsterkatalog som har utvecklats inom Serviamprojektet.

## 5.1 Relation till mönsterkatalogen

Detta avsnitt innehåller referenser till den mönsterkatalog som har skapats i Serviam (rapportnummer). Beskrivningen är på en basis av vilka mönster som dels har använts, och som dels kunde ha använts.

### 5.1.1. Juridikmönster

Arbetet i PoC har inte inkluderat juridiska aspekter. Det är ändå möjligt att resonera kring eventuella juridiska problem och synvinklar som skulle kunna komma ifråga om denna Web Service skulle användas i verkliga affärstransaktioner. Diskussionen kommer att avgränsas till samma typer av organisationer som PoC har innehållit, för att undvika att resultatet blir för abstrakt eller för långt. Som en bas används sju juridiska mönster från den mönsterkatalog som tagits fram inom ramen för projektet Serviams andra år (se figur 6).




**Figur 6: Illustration av relationen mellan de juridiska mönstren**

Fyra mönster kan komma ifråga för vårt PoC på följande sätt (lätt skuggade i figur 6):

- *Personlig integritet* : Problemet innefattar att personlig information om människor ofta används felaktigt på Internet, vilket hotar den personliga integriteten. I PoC-

fallet finns kundinformation som behöver skyddas. På ITeas sida finns kundinformation om adresser, kort, med mera. På SEBs sida finns kortrelaterad information. Rådet är att jurister ska kopplas in innan Web Services-arkitekturen skapas för att säkra att relevanta lagar följs.

- *Avtalsrätt*: Problemet är att många parter som samverkar via SOA och Web Services saknar eller har brister i sina juridiskt bindande avtal. Avtalen saknar också ofta angiven giltighetsperiod. För PoC är detta i högsta grad aktuellt. Organisationerna behöver ha ett avtal mellan sig som reglerar t.ex. tillgänglighet, uppdatering, vad Web Services får nyttjas till, etc. Det organisationerna bör göra innan SOA-arkitekturen skapas är att först undersöka om det finns standardavtal skapade av aktuella branschorganisationer; och sedan att tillsammans med jurister fastställa vilka avtal som kan bli aktuella, hur de sluts och med vilket innehåll.
- *E-handel och informationssamhällets tjänster*: Problemen är t.ex. att det är oklart hur ett erbjudande av Web Services ska hanteras med tanke på att Web Services omfattas av lagen om elektronisk handels definitioner. Gränssnittsstandarder uppfylls inte heller alltid, och lagen tolkas dessutom olika. Själva PoC handlar om elektronisk handel och betalning via Web Services. Därför är detta mönster i högsta grad aktuellt. Inblandade parter behöver analysera lagen om elektronisk handel och fastställa hur deras Web Services/SOA påverkas av den, liksom av lagen om distansavtal.
- *Elektroniska signaturer*: Problemet är att befintliga säkerhetsramverk inte tar tillräcklig hänsyn till juridiska aspekter kring t.ex. personlig integritet. De inkluderar inte heller elektroniska signaturer i tillräcklig grad. I PoC används lägsta grad av säkerhet. Hade denna Web Services använts i en verklig miljö skulle elektroniska signaturer och betydelsen av detta mönster bli betydligt större. Då skulle säkerhetsnivån behöva öka, och ett sätt att göra det är att inkludera elektroniska signaturer i arkitekturen.

### 5.1.2. Planeringsmönster

Här kommer relationer att beskrivas mellan planeringsmönster och PoC.

SOA är i grunden ett koncept som bygger på löst kopplade tjänster, implementerade i olika utvecklingsmiljöer och på olika plattformar, som kan söka efter och anropa varandra utan att någon bakomliggande programkod skall behöva ändras, det vill säga lokaliseringstransparens och implementeringstransparens. För att åstadkomma detta behövs någon form av miljö att implementera SOA tänkandet i. Det finns ett mönster som relateras till planering, att realisera SOA med Web Services.

Mönstret har använts i PoC-arbetet i och med att hela syftet med detsamma var att åstadkomma integrering mellan olika mjukvarusystem externt mellan verksamheter. Dock har de system som kopplats samman inte i huvudsak varit utvecklade i olika miljöer. Det valet gjordes av enkelhetsskäl.

### 5.1.3. Upptäcktsmönster

Här kommer relationer att beskrivas mellan upptäcktsmönstren och PoC. Beskrivningen är på en basis av vilka mönster som dels har använts, och som dels kunde ha använts.

För att kunna exponera en tjänst och i och med detta få aktörer som vill anropa tjänsten behöver denna på något sätt publiceras. För att göra det möjligt för aktörer att hitta en passande tjänst behövs någon form av sökfunktion som kan ge sökande aktörer tillräckligt detaljerad information om de tjänster som står till buds. Två mönster relateras till upptäckt: att göra en Web Service tillgänglig för andra aktörer, samt att göra en Web Service sökbar.

Följande mönster har tillämpats i arbetet med PoC:

- *Att göra en Web Service tillgänglig för andra aktörer*: I och med att en Web Service skapats i PoC har kommunikationen mellan olika arkitekturer förenklats och kommunikationen har kunnat genomföras utan manuell programmering av klasser.

Följande mönster skulle kunna ha tillämpats i arbetet med PoC:

- *Att göra en Web Service sökbar*. PoC har inte nyttjat t.ex. UDDI eller försökt göra Web Servicen användbar för andra än de två inblandade parterna. Orsaken har varit att det just har handlat om ett proof of concept, och alltså inte om en verklig situation. Däremot skulle detta drag kunna läggas till vid en utökning eller vidarearbetning av Web Servicen, och i så fall skulle mönstret bli aktuellt.

### 5.1.4. Kompositionsmönster

Här kommer relationer att beskrivas mellan kompositionsmönstren och PoC. Beskrivningen är på en basis av vilka mönster som dels har använts, och som dels kunde ha använts.

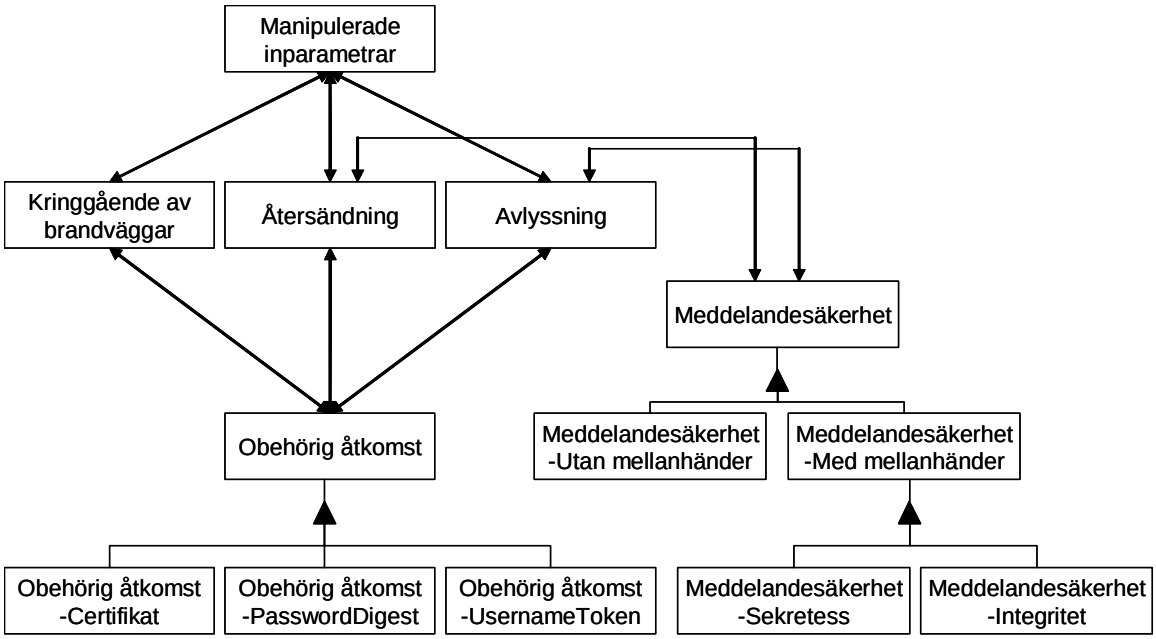
Varje Web Service är en individuell komponent. I många affärssituationer är det nödvändigt att utföra flera uppgifter för att slutföra aktiviteter. Dessa uppgifter kan implementeras i en eller flera Web Services. På grund av detta finns det ett behov av att samordna och ”koreografera” Web Services som ska utföra en uppgift/slutföra en aktivitet tillsammans. Det finns två kompositionsmönster: att skapa koreografi för en Web Service, samt att genomföra orkestrering av en Web Service. I PoC har endast koreografi-mönstret tillämpats, eftersom det har rört sig om en inter-organisatorisk Web Service. Orkestrering tillämpas endast då tjänster kontrolleras av en enda part, t.ex. inom en organisation. Vårt exempel blandar in två parter, och alltså är inte mönstret tillämpbart.

Följande mönster har tillämpats i arbetet med PoC:

- Att skapa koreografi för en Web Service – I ett B2B scenario är det nödvändigt att koordinera affärspartners uppgifter i en viss bestämd ordning. I PoC anropar parterna varandra i en viss bestämd ordning, vilket gör mönstret tillämpbart. Koordineringen av tjänsterna kontrolleras vidare av båda partner.

### 5.1.5. Säkerhetsmönster

Arbetet i PoC har inkluderat vissa säkerhetsaspekter, trots att fler hade varit att föredra (Se kapitel 4.1). Avsikten med detta kapitel är diskutera kring de säkerhetsmönster som faktiskt tillämpats i PoC samt att diskutera vilka övriga mönster som skulle ha kunnat tillämpas om den Web Services som framarbetats skulle ha använts under verkliga affärstransaktioner. Basen för diskussionen är de mönster som redovisas i mönsterkatalogen för Serviam. En översikt över säkerhetsmönstren visas i figur 7.




**Figur 7: Illustration av relationer mellan säkerhetsmönster**

Följande mönster har tillämpats i arbetet med PoC:

- Obehörig åtkomst – User name password token: En användare, i detta fall ITEA, måste på något sätt kunna bevisa att denne har rätt att utnyttja aktuell Web Service, det vill säga att autentisera sig. Det enklaste sättet att göra detta är att skicka ett användarnamn associerat med ett visst lösenord till aktuell Web Service. Om den Web Service som anropas finner en matchning mellan användarnamn och lösenord ses detta som ett bevis på att den anropande sidan har åtkomst till aktuell Web Service. Fördelen med denna lösning är att den mycket enkel. Dessvärre finns det även stora nackdelar varav de mest påfallande är att vem som helst som avlyssnar kommunikationen mellan parterna kan snappa upp användarnamn och lösenord och sedan själv anropa tjänsten utan att vara godkänd som användare av den Web Service som anropas. Det enda reella alternativet att använda User name password token är att göra detta i kombination med SSL, som ger ett insynsskydd för både autentiseringsinformation samt den data som skickas.

Följande mönster skulle kunna ha tillämpats i arbetet med PoC:

- Obehörig åtkomst – Certifikat alternativt PasswordDigest: Dessa mönster syftar till att, precis som ovanstående mönster, hantera autentisering av anropande entiteter. Skillnaden mellan dessa mönster och ovanstående mönster är att säkerheten är betydligt högre vad gäller möjligheter att komma åt användaruppgifter. Dessutom är dessa lösningar mer skalbara (Lättare att ha fler användare till en Web Service) samt fungerar fristående från SSL, vilket ju är en grundpremiss för publika Web Services som ska ha möjlighet att kopplas samman med flera olika tjänster.
- Meddelandesäkerhet: Förutom autentiseringsinformation innehåller ett SOAP-meddelande data som kan vara av mer eller mindre känslig natur. Denna data bör naturligtvis skyddas för insyn av obehöriga. Om kommunikationen mellan

anropande entitet och Web Services är direkt, det vill säga punkt till punkt, kan kommunikationen skyddas med SSL. Om det däremot finns en eller flera mellanhänder (Någon form av applikation som arbetar på SOAP-nivå och sitter mellan anropande entitet och Web Services) räcker inte SSL utan meddelandet måste skyddas på annat sätt. Det enda idag existerande sätter att göra detta är att utnyttja XML Encryption som krypterar valda delar av ett meddelande samt XML Signature som signerar valda delar av ett meddelande. Dessa tillsammans erbjuder primärt sekretess och integritet för SOAP-meddelanden som skickas via mellanhänder.

### 5.1.6. Kommunikationsmönster

Här kommer relationer att beskrivas mellan kommunikationsmönstren och PoC. Beskrivningen är på en basis av vilka mönster som dels har använts, och som dels kunde ha använts.

Att använda tjänster innefattar alltid att data med olika syfte kommuniceras mellan en eller flera aktörer via ett gemensamt gränssnitt. Det finns flera olika sätt att utforma kommunikation mellan två eller flera tjänster, beroende på antalet aktörer som medverkar, vilken art inblandade applikationer etc. Kommunikationsmönstren syftar på ett antal olika sätt att hantera kommunikation mellan tjänster beroende på vilken kommunikationsmiljö som eftersträvas. Det finns fem mönster: att kommunicera punkt-till-punkt mellan Web Services, att utnyttja en message broker, att förenkla filöverföring, att utnyttja notifiering, samt att skapa en gemensam ontologi.

Följande mönster har tillämpats i arbetet med PoC:

- *Att kommunicera punkt till punkt mellan Web Services*: Syftet med PoC var att erbjuda enkel, okomplicerad kommunikation mellan två aktörer. Det fokuserar främst fall där organisationer söker mer överblickbara lösningar som rör endast ett fåtal motparter. Detta stämmer väl in på vårt fall. Skalbarheten har inte varit ett problem här eftersom syftet har varit att testa tekniken.
- *Att skapa en gemensam ontologi*: Innan sammankopplingen gjordes mellan SEB och ITea så genomfördes en diskussion kring terminologi. Av problembeskrivningen i kapitel 4 framgår att det trots diskussionen uppkom ett mindre problem. Terminologidiskussionen kanske inte kan anses vara en full ontologi, men det är ett steg på väg dit.

Följande mönster skulle kunna ha tillämpats i arbetet med PoC:

- *Att utnyttja en message broker*: Om flera aktörer än SEB och ITea hade blandats in i PoC så hade det kunnat bli aktuellt att nyttja en message broker, i och med att syftet med dessa är att göra det möjligt för många aktörer och tjänster att samverka på ett effektivt sätt. Ju större komplexitet som råder desto svårare blir det att utnyttja punkt-till-punkt lösningar.
- *Att förenkla filöverföring*: I PoC har det inte varit aktuellt att skicka filer av olika slag. Vår Web Service skulle kunna utökas med detta drag, dock, i form av t.ex. videopresentationer av ITeas varor.
- *Att utnyttja notifiering*: Detta mönster skulle kunnat nyttjas för att notifiera ITeas angående förfrågningarna mot SEBs sida. På så vis undviks blockering av klienten medan denne väntar på en notifiering.

# 6 Slutsatser

## 6.1 Säkerhet:smässigt

De flesta utvecklingsmiljöer för Web Services säger att de även stödjer WSS. Detta stämmer i viss mån men ofta stöds inte allt i WSS utan endast vissa aspekter. Dessutom kan utvecklingsmiljöerna i sin tur stödja aspekter som ännu inte är fullt standardiserade ännu i WSS roadmap. Det förvirrar och är lite oroväckande att utvecklingsmiljöer säger sig ha stöd för saker som kommer att bli standardiserade men som ännu inte är det. Det kan innebära att olika utvecklingsmiljöer implementerar ”sin” syn på en standard, vilket i slutändan ger att det inte finns någon standard utan endast olika sätt att hantera ett och samma begrepp. Frågan blir då hur kompatibelt detta i så fall blir med andra utvecklingsmiljöer. Dessutom kan situationen påverkas av eventuella planer på framtida utvidgning av Web Services-användning och webbtjänstskapande. Det betyder att de situationer där SSL verkar fungera bäst idag inte nödvändigtvis är samma situationer det kommer att fungera bäst för i framtiden. Detsamma gäller för WS-Security.

## 6.2 Tekniskt

Det fungerar rent tekniskt med bara WSDL-filer. Men när du publicerar Web Services behöver du t.ex. UDDI-kommentarer för att beskriva tjänsten. När en organisation vill använda Web Services för affärer och samverkan kommer den inte ifrån avtalsbiten. Den Web Service som PoC innehåller måste distribueras med klientcertifikat för användning, m.m. Det finns alltså två nivåer, en teknisk och en affärsmässig. Detta betyder att vision och verklighet inte matchar ännu till 100%, eftersom det inte är så lätt som det utger sig från att vara.

## 6.3 Utvecklingsprocessen

När en Web Services utvecklas och designas kan ett antal problem och valsituationer uppstå. Dessa måste hanteras. Exempel är bestämning av processens granularitet; hur Web Services som implementerar dessa aktiviteter ska specificeras; om en top-down eller bottom-up-ansats ska användas; vilka fel- och transaktionshanteringsprocedurer som behöver definieras; vilken SOAP-implementation och applikationsserver som ska användas, vilken säkerhetsnivå som behövs, m.m. Det är inte omöjligt, men organisationer behöver vara medvetna om att utvecklingsprocessen kan komma att innebära nya typer av problem som de påträffade i denna PoC-process.

# Bilaga 1: Detaljerad specifikation av webbtjänstens operationer

WS_I: operation takeOrder.

**takeOrder**

tar information från kunden om köpet.

Inparametrar: formuläret (I form av XML dokumentet)

Svar: en text - köps bekräftelsen om alla controller är godkända, annars ett negativ svar med förklaring om problemet (möbler/kredit)

WS_SEB: operationer checkPurchase, executePurchase

**checkPurchase**

Kontrollerar att kortet som kunden vill betala med är giltigt och att beloppet för detta köp finnstillgängligt

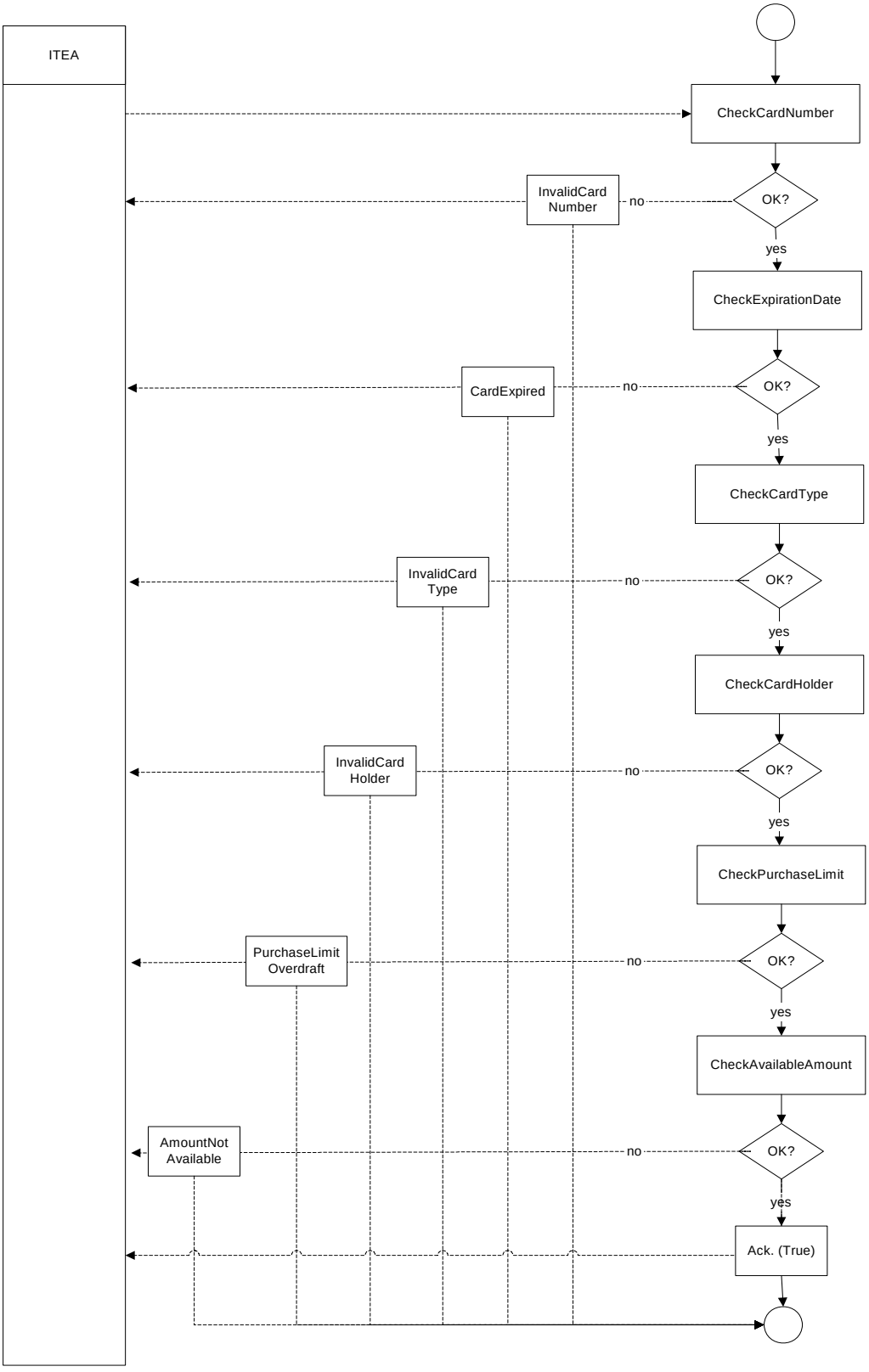
Inparametrar: cardNumber, expirationMonth, expirationYear, cardHolder, cardType (Visa/MasterCard/Diners/AmericanExpress), amount.

Svar: true om alla controller är godkända, annars false (med förklaring av problemet).

Följande felsituationer kan uppstå:

- Invalid Card Number
  - Syntaxfel eller att kort med det angivna numret existerar inte
- Card Expired
  - Kortet har gått ut
- Invalid Card Type
  - Annat än Visa, MasterCard, Diners eller AmericanExpress har angivits
- Invalid Card Holder
  - Kortet står inte registrerat på den person vars namn har angivits som kortägare
- Purchase Limit Overdraft
  - Beloppet för detta köp överskrider den gräns som är tillåtet för köp med detta kort
- Amount Not Available
  - Beloppet kvar att utnyttja på kortet är lägre än beloppet för detta köp

Exempel: Det finns 50.000 SEK tillgängligt på kortet men varje enskilt köp får ej överskrida 20.000 SEK. Ett köp på 30.000 SEK kan inte göras.



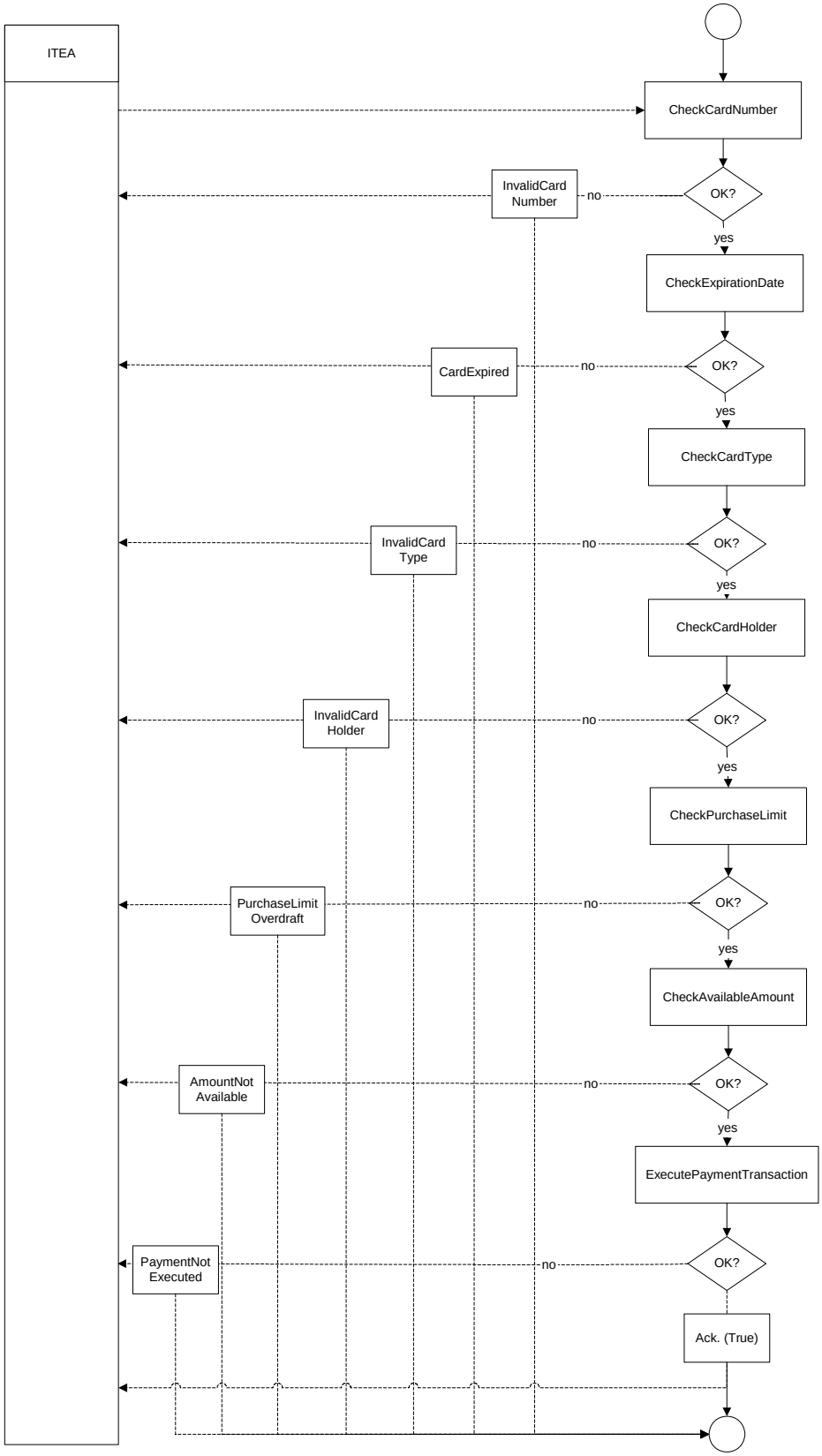

**Figur 1a: Operation checkPurchase**

**executePurchase**

Utför samma kontroller som i checkPurchase, om allt är Ok utför en betaltransaktion.

Inparametrar: Samma som hos checkPurchase
Svar: true om betaltransaktionen kunde utföras, annars false.
Felsituationer: Samma som hos checkPurchase plus följande:

- - Payment Not Executed
    - Betaltransaktionen kunde inte utföras trots att alla kontroller var OK (en annan orsak)




**Figur 1b: Operation executePurchase**

WS_L: operation processOrder.

**processOrder**

Beställer varor från leverantören.

Inparametrar: lista av varorna, med kvantitet.

Svar: true om alla controller är godkända, annars false (med information vad saknas).

# Bilaga 2: Gränssnittsdesign

Den första figuren (2.1) visar gränssnittsdesign, databärande klasser samt implementationsklasser för operationer checkPurchase och executePurchase.

Den andra figuren (2.2) visar gränssnittsdesign, databärande klasser samt implementationsklasser för samtliga operationer.

Notera att klassdiagrammet även täcker tjänster som inte demonstrerades till en början, utan dessa tjänster utvecklades utöver det som överenskommits. Gränssnitts- och dataklasser för dessa tjänster presenteras därför efter figurerna. Klasserna är: CardData, CardType, InvoicePaymentInterface och PurchaseLimitData.

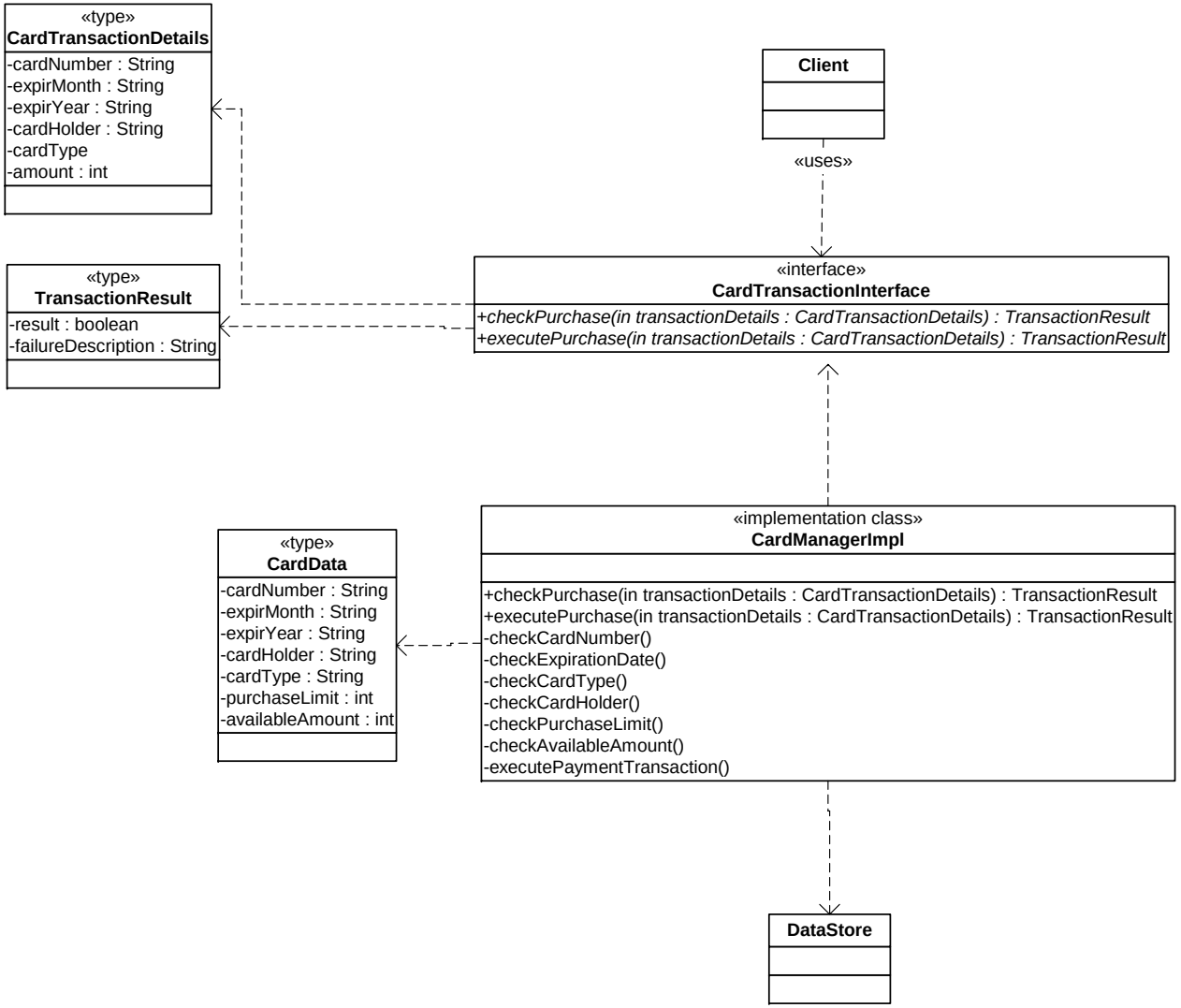

«type»
CardTransactionDetails
-cardNumber : String
-expirMonth : String
-expirYear : String
-cardHolder : String
-cardType
-amount : int
Client
«uses»
«type»
TransactionResult
-result : boolean
-failureDescription : String
«interface»
CardTransactionInterface
+checkPurchase(in transactionDetails : CardTransactionDetails) : TransactionResult
+executePurchase(in transactionDetails : CardTransactionDetails) : TransactionResult
«type»
CardData
-cardNumber : String
-expirMonth : String
-expirYear : String
-cardHolder : String
-cardType : String
-purchaseLimit : int
-availableAmount : int
«implementation class»
CardManagerImpl
+checkPurchase(in transactionDetails : CardTransactionDetails) : TransactionResult
+executePurchase(in transactionDetails : CardTransactionDetails) : TransactionResult
-checkCardNumber()
-checkExpirationDate()
-checkCardType()
-checkCardHolder()
-checkPurchaseLimit()
-checkAvailableAmount()
-executePaymentTransaction()
DataStore

Fig 2.1

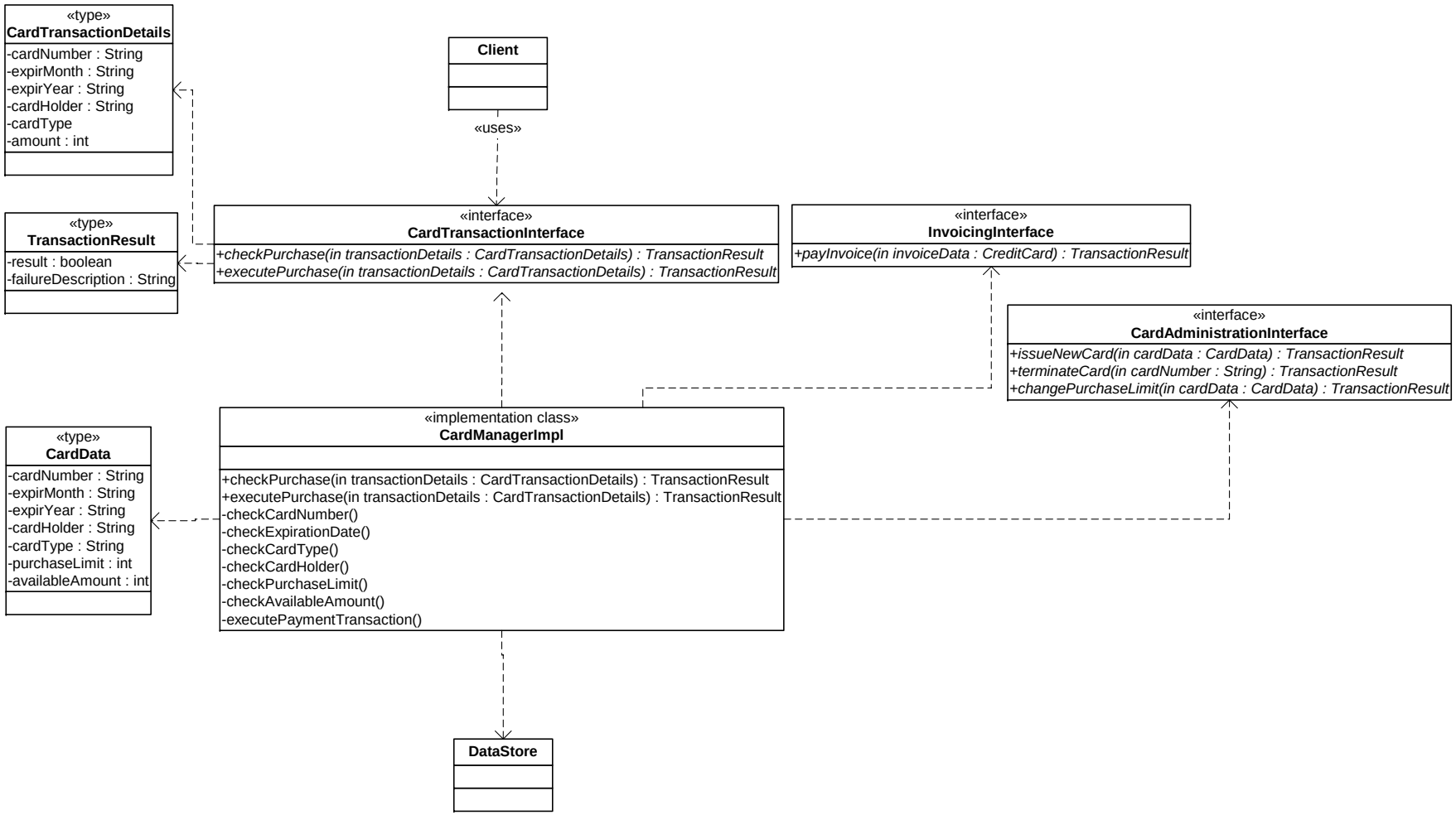

«type»
CardTransactionDetails
-cardNumber : String
-expirMonth : String
-expirYear : String
-cardHolder : String
-cardType
-amount : int
Client
«uses»
«type»
TransactionResult
-result : boolean
-failureDescription : String
«interface»
CardTransactionInterface
+checkPurchase(in transactionDetails : CardTransactionDetails) : TransactionResult
+executePurchase(in transactionDetails : CardTransactionDetails) : TransactionResult
«interface»
InvoicingInterface
+payInvoice(in invoiceData : CreditCard) : TransactionResult
«interface»
CardAdministrationInterface
+issueNewCard(in cardData : CardData) : TransactionResult
+terminateCard(in cardNumber : String) : TransactionResult
+changePurchaseLimit(in cardData : CardData) : TransactionResult
«type»
CardData
-cardNumber : String
-expirMonth : String
-expirYear : String
-cardHolder : String
-cardType : String
-purchaseLimit : int
-availableAmount : int
«implementation class»
CardManagerImpl
+checkPurchase(in transactionDetails : CardTransactionDetails) : TransactionResult
+executePurchase(in transactionDetails : CardTransactionDetails) : TransactionResult
-checkCardNumber()
-checkExpirationDate()
-checkCardType()
-checkCardHolder()
-checkPurchaseLimit()
-checkAvailableAmount()
-executePaymentTransaction()
DataStore

Fig 2.2

**CreditCardInformation**

```
/*
 * CreditCardInformation.java
 *
 * Created on den 4 april 2005, 19:26
 */

package serviam.cardservice;

/**
 *
 * @author  s70464
 */
public class CardData {
    private String cardNumber;
    private String expirMonth;
    private String expirYear;
    private String cardHolder;
    private String cardType;
    private int purchaseLimit; // Limit for single purchase
    private int credit;   // The total credit amount approved
    private int availableAmount;  // = credit - all purchases; initially availableAmount = credit

    public String getCardNumber(){
        return this.cardNumber;
    }

    public String getExpirMonth(){
        return this.expirMonth;
    }

    public String getExpirYear(){
        return this.expirYear;
    }

    public String getCardHolder(){
        return this.cardHolder;
    }

    public String getCardType(){
        return this.cardType;
    }

    public int getPurchaseLimit(){
        return this.purchaseLimit;
    }

    public int getCredit(){
        return this.credit;
    }

    public int getAvailableAmount(){
        return this.availableAmount;
    }

    public void setAvailableAmount(int amount){
```

```
        this.availableAmount = amount;
    }

     public void setPurchaseLimit(int amount){
        this.purchaseLimit = amount;
    }

     public void setCredit(int credit){
        this.credit = credit;
    }

    /** Creates a new instance of CreditCardInformation */
    public CardData() {
    }

    public CardData(String cardNumber, String expirMonth, String expirYear, String cardHolder, String
cardType, int purchaseLimit, int credit) {

         this.cardNumber = cardNumber;
         this.expirMonth = expirMonth;
         this.expirYear = expirYear;
         this.cardHolder = cardHolder;
         this.cardType = cardType;
         this.purchaseLimit = purchaseLimit;
         this.credit = credit;
         this.availableAmount = credit;

    }

    public CardData(CardData card){

         this.cardNumber = card.cardNumber;
         this.expirMonth = card.expirMonth;
         this.expirYear = card.expirYear;
         this.cardHolder = card.cardHolder;
         this.cardType = card.cardType;
         this.purchaseLimit = card.purchaseLimit;
         this.credit = card.credit;
         this.availableAmount = card.availableAmount;

    }

}
```

**CardType**

```
/*
 * CardType.java
 *
 * Created on den 4 april 2005, 10:26
 */

package serviam.cardservice;

/**
 *
 * @author  S70464
 */
public class CardType {

    public final static String VISA = "0";
    public final static String MASTERCARD = "1";
    public final static String DINERS = "2";
    public final static String AMEXPRESS = "3";


}
```

**InvoicePaymentInterface**

```
/*
 * InvoicePaymentInterface.java
 *
 * Created on den 7 april 2005, 19:43
 */

package serviam.cardservice;

/**
 *
 * @author  s70464
 */
public interface InvoicePaymentInterface {

    // payInvoice - increases the amount of money available on the card
    public TransactionResult payInvoice ( CardTransactionDetails transDetails);


}
```

**PurchaseLimitData**

```
/*
 * PurchaseLimitData.java
 *
 * Created on den 7 april 2005, 18:41
 */

package serviam.cardservice;

/**
 *
 * @author  s70464
 */
public class PurchaseLimitData {

    private String cardNumber;
    private int purchaseLimit;

    /** Creates a new instance of PurchaseLimitData */
    public PurchaseLimitData() {
    }

    public PurchaseLimitData(String cardNumber, int limit){
        this.cardNumber = cardNumber;
        this.purchaseLimit = limit;
    }

    public String getCardNumber(){
        return this.cardNumber;
    }

    public int getPurchaseLimit() {
        return this.purchaseLimit;
    }
    }
```

# Bilaga 3: Säkerhetsrekommendationer

**Lösningsförslag 1**

Detta lösningsförslag fokuserar i första hand på att komma fram till en relativt enkel lösning som dessutom passar det utvecklingsverktyg (WebSphere) som kommer att användas för utveckling av tjänsten. Absoluta krav när det gäller säkerhet för hela webbtjänsten är att användaren, i detta fall ITEA, skall kunna autentisera sig emot tjänsten och få tillgång till denna. Dessutom måste det kontokortsnummer som skickas till tjänsten från ITEA skyddas så att detta inte skall kunna läsas under transporten mellan ändnoderna, det vill säga ITEA och SEB. Lösningen strävar efter att, i mån av möjlighet, inte använda sig av äldre säkerhetsmekanismer, hanterande transportsäkerhet, utan istället fokusera på nyare lösningar som kan hantera meddelandesäkerhet, alltså WSS.

Grundarkitekturen för interaktionen ser ut som nedan i figur 3.1:

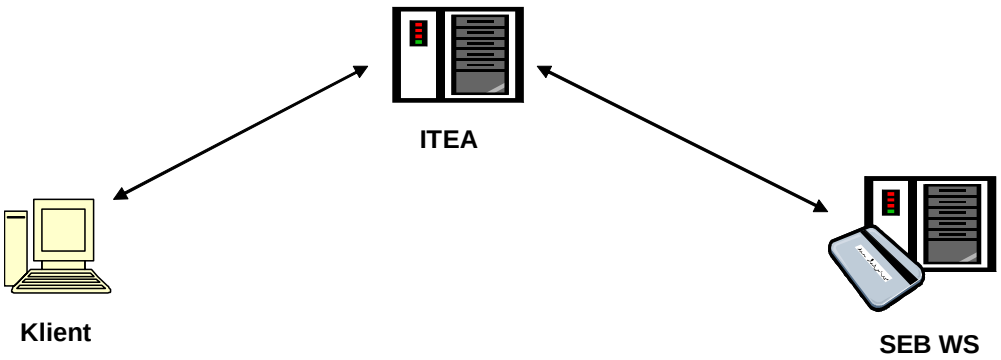



Fig.3.1: Grundarkitektur för PoC

Grundprocessen kan sedan delas in i två delar:

1. Kundens registrering hos ITEA:
    a. Kunden skickar in grundläggande uppgifter till ITEA via en oskyddad kanal (Pnr, Levadress, Kontokortsnummer, Användarnamn)
    b. ITEA genererar ett lösenord som i kombination med användarnamn kan autentisera en användare
2. Kundens köp hos ITEA
    a. Kunden loggar in till sitt konto och gör sin beställning
    b. ITEA skickar ett meddelande till SEB innehållande autentiseringsuppgifter samt kontokortsnummer
    c. SEB ger ett svar till ITEA

Det finns två olika delar av meddelandet som måste skyddas, lösenordet samt kontokortsnumret.

**Lösenordet**
WSS gör det möjligt att specificera lösenord (UsernameToken) för att autentisera en användare, lösenordet i klartext måste naturligtvis vara känt på båda sidor. Det absolut enklaste sättet är naturligtvis att skicka lösenordet i klartext i meddelandet. Detta kanske dock inte är jättebra om SEB inte vill att vem som helst skall kunna utnyttja tjänsten. Detta alternativ går alltså bort.

Dock är det möjligt, med relativt enkla medel, att kryptera lösenordet i meddelandet. Vad som används är en slags sammanslagning (Username PasswordDigest) som utnyttjar en hash bestående av en tidsstämpel, ett slumptal samt lösenordet. Detta sätts samman och skickas, packas upp på motstående sida och lösenordet kan sedan utvärderas.

Scenario
1. ITEA anropar SEB genom ett SOAP-meddelande.
2. SEB tittar i huvudet på SOAP-meddelandet, läser användarnamnet och dekrypterar lösenordet.
3. Om det finns en matchning fortsätter ärendet, annars skickas en reject tillbaka till ITEA.
4. SEB hämtar uppgift (Boolean eller Int?) i sitt källsystem.
5. SEB returnerar ett svar på SOAP-meddelandet.

**Fördelar**
- Vi slipper hantera certifikat etc. som är ett önskemål från SEB.
- Vi får ett hyfsat skydd mot replay- och man in the middle attacker (Detta skydd kan göras olika starkt beroende på hur noggrann man är med att specificera spektrat för tidsstämplar, loggning av slumpvärden osv, dessutom kan ju lösenordet i sig vara en egen digest etc.)
- Lösningen är enkel och bör inte vara speciellt komplex att implementera.

**Nackdelar**
- Lösenord är alltid känsligt och går att knäcka
- WebSphere menar på att PasswordDigest inte stöds, men varianter av detta kan ändå åstadkommas (Base64 kryptering)
- Om fler noder blandas in kan det bli svårt att skala upp lösningen

**Kontokortsnumret**
Det viktigast för kontokortsnumret är att ingen utomstående betraktare kan läsa detta mellan de båda ändnoderna, alltså att upprätthålla sekretess. Att skydda meddelandets integritet känns inte lika viktigt. De enda sättet att erhålla sekretess är att använda sig av kryptering, alltså XML-Encryption för WSS. Det man gör (på enklaste sätt) är att ge en referens-URI i säkerhetsblocket som pekar mot kontonumret i kroppen på meddelandet. I detta case skulle kryptering kunna ske symmetriskt då ITEA och SEB kan erhålla varsin nyckel som inte behöver skickas med i meddelandet.

**Fördelar**
- Enklast möjliga sätt att få till en bra kryptering med hjälp av symetriska nycklar

**Nackdelar**

- Asymmetriska nycklar blir svårare att skala upp
- Lösningen är inte helt följande WSS genom ovanstående. WSS standardvariant på detta är att skapa en ”session-key” med hjälp av publika nycklar och certifikat.

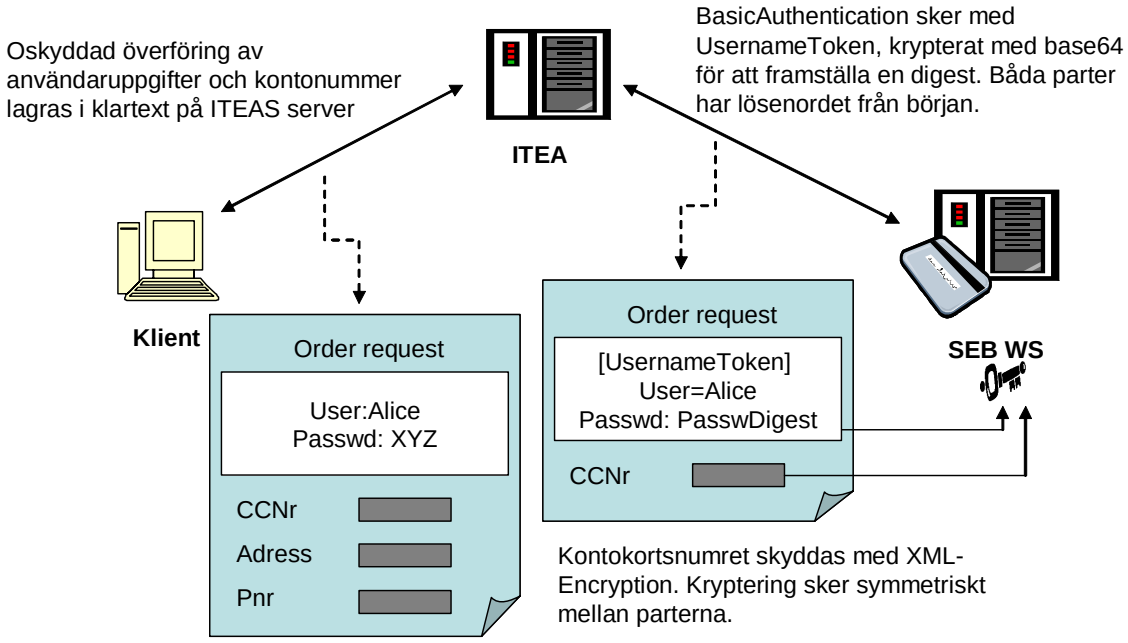



Fig 3.2: Lösningsförslag 1

**Lösningsförslag 2**

Detta lösningsförslag fokuserar mer än lösningsförslag ett mot ett scenario som täcker upp fler aspekter än de allra nödvändigaste när det kommer till att säkra utbytet av meddelanden mellan kunden, ITEA och SEB.

De krav på säkerhet som finns i denna lösning är lite hårdare än i lösningsförslag 1 och innefattar även en säker interaktion mellan klient och ITEA, inta bara ITEA till SEB som är fallet i lösningsförslag 1. Förslaget bygger helt och hållet på WSS i kombination med WebSphere och har extremt stora likheter med det case som beskrivs i användarmanualen för WebSphere 6.0, sid 451, dock finns det mer detaljer och förklaringar i detta lösningsförslag. Lösningsförslag 2 kräver hantering av certifikat etc. som inte önskats av SEB.

Klienten autentiserar sig mot ITEA med ett användarnamn och ett lösenord. Klienten skickar även med sitt kontokortsnummer till ITEA. ITEA autentiserar kunden men skickar vidare kortnumret till SEB för kontroll. ITEA tittar alltså aldrig på kontokortsnumret utan vidarebefordrar detta direkt till SEB. Det finns ju ingen anledning för ITEA att titta på detta nummer då SEB hanterar alla betalningsförfaranden. SEB tar alltså hand om betalningsprocedurer och ITEA fokuserar endast på att leverera de varor som beställs.

**Lösenordet**
Lösenordet är i detta fall en binär nyckel (BinarySecurityToken) av typen X.509 som bygger på asymmetrisk kryptering.

**Signatur**
För att få ihop autentiseringen behövs även en digital signatur som visar att klienten har den rätta privata nyckeln

**Kryptering**
Meddelandet krypteras i två skikt, där ITEA endast kan läsa de uppgifter som har med ordern att göra medan SEB endast kan läsa kortnumret.

Följande figur tillsammans med beskrivning kan få läget att bli lite klarare:

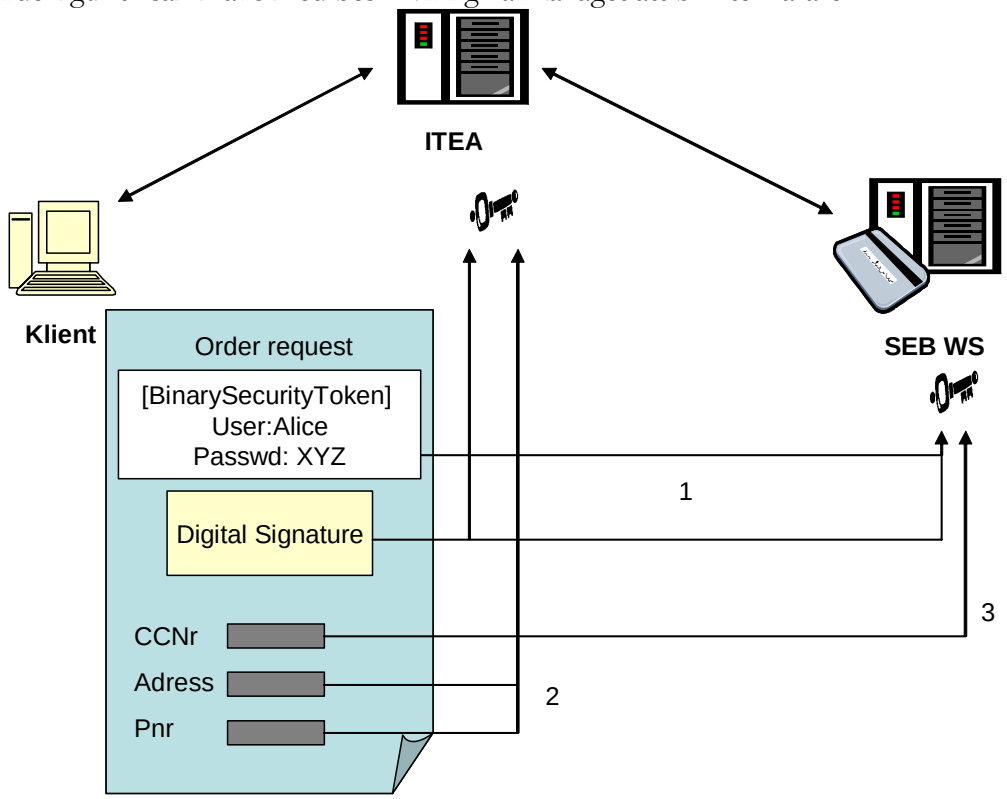



Fig 3.3: Lösningsförslag 2

**Scenario**

Klienten skickar en request om att köpa någonting via Internet. Request innehåller kontokortsnummer, leveransadress och produktnummer.

- Kontokortsnumret skall endast vara synligt för SEB.
- Leveransadress och produktnummer skall endast vara synligt för ITEA.
- Kryptering av ovanstående uppgifter kan göras med hjälp av *Key Wrapping*, alltså att avsedda delar av meddelandet krypteras med en gemensam symmetrisk nyckel som därefter krypteras med en asymmetrisk nyckel. Avsändaren krypterar alltså den

symmetriska nyckeln med mottagarens publika nyckel. Mottagaren kan sedan dekryptera den asymmetriska nyckeln med sin egen privata nyckel och få fram vilken symmetrisk nyckel som gäller och därefter dekryptera ”sina” delar av meddelandet.

- I detta case finns alltså en grundläggande skillnad mot lösningsförslag 1 då SEB hanterar betalningen direkt. Dock behöver SEB ändå kunna returnera ett boolean-värde som informerar om klienten har pengar eller ej. Dock är det SEB som hanterar betalningen, inte ITEA.
- Klienten kan dessutom signera meddelandet, med exempelvis X.509 certifikat, vilket gör att klienten kan bevisa att denne är den som har den privata nyckeln (Om klienten signerar meddelandet med sin privata nyckel och mottagaren kan använda den publika nyckeln i X.509 certifikatet styrker detta att mottagaren är den denne utger sig för att vara). Signeringen gör dessutom att meddelandets integritet kan styrkas hos mottagaren så länge denne validerar signaturen enligt ovan.

    1. Båda parter kan autentisera klienten med hjälp av den symmetriska nyckeln. Signeringen av meddelandet styrker detta ytterligare.
    2. Endast adress och produktnummer kan dekrypteras av ITEA
    3. Endast kontokortsnummer kan dekrypteras av SEB.

**Fördelar**

- Sekretess och integritet för meddelande kan upprätthållas trots multipla mottagare av ett SOAP-meddelande. Vi når alltså meddelandesäkerhet istället för transportsäkerhet och kan hantera mellanhänder (ITEA).
- Användning av certifikat gör det enklare att skala upp lösningen.
- Vi följer WSS och utnyttjar alla delkomponenter i någon omfattning.
- Via signering kan vi även uppvisa oavvislighet

**Nackdelar**

- Någon form av portaltjänst behövs för att det ska vara möjligt att skicka ett meddelande av den typ som beskrivs ovan. Det räcker alltså inte med bara en klient.
- Betydligt mer komplext än i lösningsförslag 1.
- Vi måste hålla koll på certifikat mellan klient, ITEA och SEB.
- ITEA måste utökas med ny funktionalitet
- Osäkert om hur olika utvecklingsmiljöer (WebSphere och JavaBeans) funkar ihop med avseende på säkerhet. Dock vore detta intressant att studera.

# Bilaga 4: Orkestreringslogik för WS

```
================================================================
= -->
        <!-- ORCHESTRATION LOGIC                                  -->
        <!-- Set of activities coordinating the flow of messages across the    -->
        <!-- services integrated within this business process                  -->
        <!--
================================================================
= -->

<sequence name="main">
                <!-- Receive input from requestor.
        Note: This maps to operation defined in IteaBPEL.wsdl
        -->
                <receive name="receiveInput" partnerLink="client" portType="tns:IteaBPEL"
operation="initiate" createInstance="yes" variable="input"/>
                <!-- Asynchronous callback to the requester.
        Note: the callback location and correlation id is transparently handled
        using WS-addressing.
        -->
                <scope name="scope-2"><sequence><assign name="assign-1">
                                        <copy>
                                                <from variable="input" part="payload"
query="/tns:IteaBPELRequest/tns:inputCustomer"></from>
                                                <to variable="checkCustomerInput"
part="parameters" query="/tns:checkCustomer/tns:c"/>
                                        </copy>
                                </assign>
                                <invoke name="invoke-2" partnerLink="ITea1"
portType="tns:BPELWSSoap" operation="checkCustomer" inputVariable="checkCustomerInput"
outputVariable="checkCustomerOutput"/>
                        </sequence>
                </scope>
                <scope name="scope-1"><sequence><assign name="assign-2">
                                        <copy>
                                                <from variable="checkCustomerOutput"
part="parameters" query="/tns:checkCustomerResponse/tns:checkCustomerResult"></from>
                                                <to variable="calculateSumInput" part="parameters"
query="/tns:calculateSum/tns:c"/>
                                        </copy>
                                        <copy>
                                                <from variable="input" part="payload"
query="/tns:IteaBPELRequest/tns:inputPurchase"></from>
                                                <to variable="calculateSumInput" part="parameters"
query="/tns:calculateSum/tns:pi"/>
                                        </copy>
                                </assign>
                                <invoke name="invoke-1" partnerLink="ITea1"
portType="tns:BPELWSSoap" operation="calculateSum" inputVariable="calculateSumInput"
outputVariable="calculateSumOutput"/>
</sequence>
</scope>
```

# Bilaga 5: Kodexempel för säkerhet

I den här filen konfigurerar man vad man vill ha för säkerhet: ibm-webservices-ext.xmi

```
 <wsDescExt xmi:id="WsDescExt_1115814043544" wsDescNameLink="UPCardTransactionService">
  <pcBinding xmi:id="PcBinding_1115814043544" pcNameLink="UPCardTransaction">
   <serverServiceConfig xmi:id="ServerServiceConfig_1115815808274">
    <securityRequestConsumerServiceConfig
xmi:id="SecurityRequestConsumerServiceConfig_1115815808274">
     <caller xmi:id="Caller_1115815907717" name="UP_CallerPart" part="" uri=""
localName="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-
1.0#UsernameToken"/>

     <requiredSecurityToken xmi:id="RequiredSecurityToken_1115815907717" name="UP_SecurityToken"
uri="" localName="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-
1.0#UsernameToken" usage="Required"/>

    </securityRequestConsumerServiceConfig>
   </serverServiceConfig>
  </pcBinding>
 </wsDescExt>
```

Och i den här hur den implementeras ibm-webservices-bnd.xmi

```
 <wsdescBindings xmi:id="WSDescBinding_1115814043414"
wsDescNameLink="UPCardTransactionService">
  <pcBindings xmi:id="PCBinding_1115814043414" pcNameLink="UPCardTransaction">
   <securityRequestConsumerBindingConfig
xmi:id="SecurityRequestConsumerBindingConfig_1115815982224">
    <tokenConsumer xmi:id="TokenConsumer_1115815982224"
classname="com.ibm.wsspi.wssecurity.token.UsernameTokenConsumer" name="UP_TokenConsumer">

     <valueType xmi:id="ValueType_1115815982224" localName="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken" uri=""
name="Username Token"/>

     <jAASConfig xmi:id="JAASConfig_1115815982224"
configName="system.wssecurity.UsernameToken"/>
     <partReference xmi:id="PartReference_1115815982224" part="UP_SecurityToken"/>
    </tokenConsumer>
   </securityRequestConsumerBindingConfig>
  </pcBindings>
 </wsdescBindings>
```

# Bilaga 6: Javakod för Web Service-gränssnittet

Följande kod beskriver Javagränssnittet för den framtagna Web Servicen. Detta inkluderar även dataklasserna TransactionResult och CardTransactionDetails.

**CreditCardInterface**

```
/*
 * CreditCardInterface.java
 *
 * Created on den 4 april 2005, 10:09
 */

package serviam.cardservice;

/**
 *
 * @author  S70464
 *
 */
public interface CardTransactionInterface extends java.rmi.Remote{

    // checkPurchase - checks whether the credit card is valid and the purchase amount is available
    public TransactionResult checkPurchase ( CardTransactionDetails transDetails);

    // executePurchase - makes the same checks as checkPurchase and if everything is OK, executes the
purchase transaction
    public TransactionResult executePurchase ( CardTransactionDetails transDetails);

}
```

**TransactionResult**

```
/*
 * TransactionResult.java
 *
 * Created on den 4 april 2005, 09:53
 */

package Serviam0525.cardservice;

/**
 *
 * @author  S70464
 */

/**
 * TransactionResult holds the result of authorisation and purchase transactions.
 * if the transaction was successfull: result = true (failureDescription is not used)
 * if the transaction failed: result = false,
 *                          failureDescription - describes the reason for the transaction failure
 */

public class TransactionResult {

        private boolean result;
        private String failureDescription;

        public boolean getResult(){
            return this.result;
        }

        public String getFailureDescription(){
            return this.failureDescription;
        }

    /** Creates a new instance of TransactionResult */
    public TransactionResult(boolean result, String failureDescription){
        this.result = result;
        this.failureDescription = failureDescription;
    }
}
```

**CardTransactionDetails**

```
/*
 * CardTransactionDetails.java
 *
 * Created on den 4 april 2005, 09:48
 */

package Serviam0525.cardservice;

/**
 *
 * @author  S70464
 * Class holding the parameters to be sent along with a card transaction
 */
public class CardTransactionDetails {

        private String cardNumber;
        private String expirationMonth;
        private String expirationYear;
        private String cardHolder;
        private String cardType;
        private int amount;

        public String getCardNumber() {
            return this.cardNumber;
        }

        public String getExpirationMonth() {
            return this.expirationMonth;
        }

        public String getExpirationYear() {
            return this.expirationYear;
        }

        public String getCardHolder(){
            return this.cardHolder;
        }

        public String getCardType(){
            return this.cardType;
        }

        public int getAmount(){
            return this.amount;
        }


    /** Creates a new instance of CardTransactionDetails */
        public CardTransactionDetails() {
        }

        public CardTransactionDetails(  String cardNumber,
                                        String expirationMonth,
                                        String expirationYear,
                                        String cardHolder,
```

```
                                String cardType,
                                int amount) {

        this.cardNumber = cardNumber;
        this.expirationMonth = expirationMonth;
        this.expirationYear = expirationYear;
        this.cardHolder = cardHolder;
        this.cardType = cardType;
        this.amount = amount;

    }
        // Used for invoice payment transaction
        public CardTransactionDetails(String cardNumber,int amount) {
            this(cardNumber,null,null,null,null,amount);
        };

}
```

# Bilaga 7: Utveckling av klienten i ITea-SEB projektet

Till att börja med installerades J2EE 1.4 plattformen och Sun:s utvecklingspaketet för Web Services, Java Web Services Developer Pack 1.5 (JWSDP). JWSDP innehåller allt som krävs för att kunna skapa och anropa webbtjänster med J2EE plattformen. Det innehåller bl a API för XML processering (JAXP), XML register (JAXR) och XML baserad Remote Procedure Calls (JAX-RPC). Utöver dessa API erbjuder JWSDP också bl a stöd för Web Services Security 1.0, XML Digital Signatures 1.0 och WS-I Attachments. Ett antal utvecklings verktyg medföljer också paketet vilket underlättar skapande av både webbtjänster och klienter som anropar tjänster.

Med verktyget *wscompile* som medföljer JWSDP går det att utifrån en WSDL fil automatiskt generera klient proxyn som sköter serialisering av meddelanden till och från SOAP. Denna proxy abstraherar bort mycket av den underliggande komplexiteten vid anrop av webbtjänster. En användare behöver således inte i normalfallet tänka på hur serialiseringen sker utan kan på en högre abstraktionsnivå anropa en operation hos en webbtjänst som om det logiskt sett vore en lokal operation.

Verktyget *wscompile* behöver få kännedom om var SEB:s WSDL fil finns. Utifrån denna WSDL fil autogenererar *wscompile* klientens proxy. SEB:s WSDL fil finns på följande URL: http://swp9.vv.sebank.se/cgi-bin/pts3/wsh/sam/wsdl/serviam/cardservice/usernamepassword/UPCardTransaction.wsdl. Denna URL anges i en konfigurations fil som är skriven med XML syntax. Konfigurationsfilen anges sedan som inparametar till *wscompile* när verktyget exekveras. Konfigurationsfilen innehållandes URL:en till WSDL filen döptes till config.xml och såg ut på följande sätt:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jaxrpc/ri/config">
        <wsdl location="http://swp9.vv.sebank.se/cgi-
                bin/pts3/wsh/sam/wsdl/serviam/cardservice/
                usernamepassword/UPCardTransaction.wsdl" packageName="itea.logic.SEB"/>
</configuration>
```

Som vi ser innehåller wsdl elementet två attribut, dels URL:en till WSDL filen och dels den logiska paket strukturen för proxyns klassfiler (itea.logic.SEB).

Vi behöver dessutom tala om för *wscompile* att klientapplikationen ska anropa webbtjänsten med WS-Security och specifikt med Username Token tekniken. Denna säkerhetsinformation ska skrivas i en säkerhetskonfigurationsfil som också anges som inparameter till *wscompile* då klient proxyn automatiskt genereras. Denna fil döps till security.xml och ser ut på följande vis:

```
  <xwss:JAXRPCSecurity
xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
            <xwss:Service>
             <xwss:SecurityConfiguration dumpMessages="true">
            <xwss:UsernameToken name="serviam"
                                        password="lösenord" digestPassword="false"/>
             </xwss:SecurityConfiguration>
  </xwss:Service>
  <xwss:SecurityEnvironmentHandler>
            itea.logic.SEB.ClientSecurityEnvironmentHandler
  </xwss:SecurityEnvironmentHandler>
</xwss:JAXRPCSecurity>
```

Säkerhetsinformationen byggs således in i klient proxyn när denna skapas. Elementet UsernameToken i filen ovan innehåller användarnamnet och lösenordet som ska läggas i SOAP huvudet innan det skickas iväg till webbtjänsten. Lösenordet skickas i det här fallet i klartext till tjänsten vilket inte är föredra i praktiken, utan man bör se till så att lösenordet krypteras. Det krävs också en sk SecurityEnvironmentHandler klass för att få säkerheten att fungera korrekt och framförallt för att kunna kompilera klassfilerna. Denna klass returnerar lösenordet till klientapplikationen i de fall då lösenordet inte hårdkodas in i säkerhetskonfigurationsfilen, i vårt fall hårdkodades just lösenordet i filen, men SecurityEnvironmentHandler klassen måste trots allt finnas för att kunna kompilera klienten korrekt.

*Wscompile* körs alltså från en vanlig kommandoprompt och nedan står exekveringskommandot med samtliga parametrar.

```
wscompile –security security.xml -verbose -gen -d classes -s src
-keep etc/config.xml
```

De olika flaggorna i kommandot betyder följande:

-*security* anger att en säkerhetskonfigurationsfil följer. Namnet på filen är i detta fall security.xml.
-*verbose* innebär att verktyget ska skriva ut meddelanden på prompten om vad som sker.
-*gen* innebär att klient artefakter ska genereras.
-*d* specifierar vart de kompilerade klassfilerna ska placeras. I vårt fall läggs de i en mapp med namnet ”classes”.
-*s* specifierar vart källkodsfilerna ska placeras. I vårt fall läggs de i en mapp med namnet ”src”.
-*keep* talar om för verktyget att källkodsfilerna inte ska tas bort efter att klassfiler har kompilerats från dem.

Slutligen anges vart konfigurationsfilen med WSDL URL:en finns. I vårt fall lades config.xml i en mapp med namnet ”etc”.

Efter att detta kommando har körts ligger följande källkodsfiler (inklusive vår egendefinierade ClientSecurityEnvironmentHandler klass), som automatiskt genererats av *wscompile*, i mappen ”src”.

Fig. 7.1. Källkodsfiler genererade av wscompile.

För att testa proxyn kan vi skriva en liten klient klass som instansierar proxyn och anropar SEB:s tjänst med relevanta inparametrar. En av operationerna i SEB:s tjänst heter checkPurchase. Denna operation tar in ett objekt av klassen CardTransactionDetails innehållandes transaktionsinformation. Operationen checkPurchase kontrollerar transaktionsinformation och returnerar ett objekt av klassen TransactionResult. Detta TransactionResult objekt innehåller attributet ”result” av booleskt värde som är satt till true om transaktionsinformationen är korrekt, annars false. För att instansiera proxyn, anropa checkPurchase operationen och skriva ut värdet på resultatet använder testklient klassen följande kodsnutt:

```
CardTransactionDetails ctd=new CardTransactionDetails(“1111”, “08”, “15”, “Kalle Jonsson”,
“VISA”, 1000);

UPCardTransaction ws = new UPCardTransactionService_Impl().getUPCardTransaction();

TransactionResult tr=new TransactionResult();
tr=ws.checkPurchase(ctd);
System.out.println(“Returvärde: ”+tr.getResult());
```

I exemplet ovan har vi hårdkodat transaktionsdata som skickas till tjänsten. SOAP meddelandet som skickas iväg ser ut enligt följande:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
        xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:ns0="http://cardservice.serviam"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <env:Header>
                <wsse:Security xmlns:wsse="http://docs.oasis-
                        open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
                        env:mustUnderstand="1">
                        <wsse:UsernameToken>
                                <wsse:Username>serviam</wsse:Username>
                                <wsse:Password>****</wsse:Password>
                                <wsse:Nonce
                                        EncodingType="http://docs.oasis-
                                        open.org/wss/2004/01/oasis-200401-wss-soap-
                                        message-security-1.0#Base64Binary">
                                                furgepywCLJVBxCVmZS5fMnX
                                </wsse:Nonce>
                                <wsu:Created
                                        xmlns:wsu="http://docs.oasis-
                                        open.org/wss/2004/01/oasis-200401-wss-
                                        wssecurity-utility-1.0.xsd">
                                        2005-05-25T 17:27:07Z
                                </wsu:Created>
                        </wsse:UsernameToken>
                </wsse:Security>
        </env:Header>
        <env:Body>
                <ns0:executePurchase>
                        <transDetails>
                                <amount>1000</amount>
                                <cardHolder>Kalle Jonsson</cardHolder>
                                <cardNumber>1111</cardNumber>
                                <cardType>VISA</cardType>
                                <expirationMonth>08</expirationMonth>
                                <expirationYear>15</expirationYear>
                        </transDetails>
                </ns0:executePurchase>
        </env:Body>
</env:Envelope>
```

Hela paketet (itea logic SEB) med alla nödvändiga klassfiler för proxyn infördes i en av EJB komponenterna i ITea systemet. Vid beställningsförfarandet i ITea lades det till ett anrop till SEB tjänsten enligt liknande tillvägagånssätt som med testklienten genom att instansiera proxyn. Webbsajten utökades även med html sidor för att visa felmeddelanden från SEB tjänsten. Slutligen installerades systemet på en JBoss applikationsserver. Självfallet behöver man se till så att alla nödvändiga jar-filer för både grundläggande JWSDP funktionalitet och säkerhetsfunktionalitet kopieras till rätt katalog i applikationsservern så att dessa jar-filer hittas vid exekvering av applikationen som har deployats på servern. ITea sajten kan nås på iteaserviam.dsv.su.se.

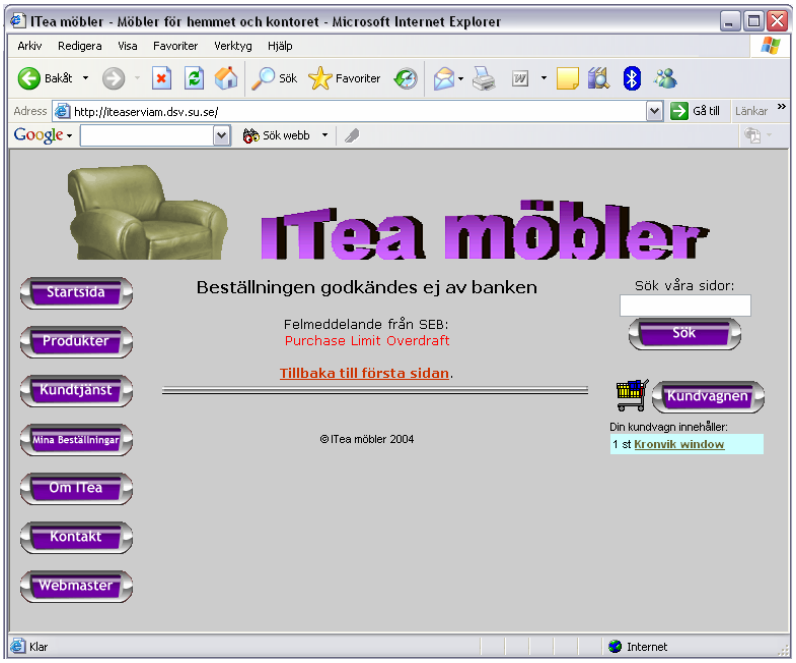



Fig. 7.2. SEB tjänsten godkände inte kundens kontokortsuppgifter varpå ett felmeddelande visas på skärmen.

# Bilaga 8: Jar-filer som måste anges i classpath

Alla nödvändiga jar-filer måste anges i classpath vid kompilering och exekvering av klientapplikationer som använder sig av Java Web Services Developer Pack 1.5's (JWSDP) API. Nedan följer en sammanställning av de jar-filer som krävs för att kunna exekvera en klient både med och utan säkerhetsfunktionalitet, samt i vilka kataloger de finns. Jar-filerna finns i respektive katalogs lib-katalog. Om JWSDP 1.5 installeras på C:\ hårdisken skulle tex. sökvägen till katalogen jaxrpc vara C:\jwsdp-1.5\jaxrpc\ och nödvändiga jar-filer från denna katalog skulle finnas i C:\jwsdp-1.5\jaxrpc\lib\.

| ***Katalog: jaxrpc*** | ***Katalog: xws-security (för WS-Security)*** | ***Katalog: jwsdp-shared*** |
|---|---|---|
| jaxrpc-api.jar | keyexport.jar | namespace.jar |
| jaxrpc-impl.jar | pkcs12import.jar | jax-qname.jar |
| jaxrpc-spi.jar | security-plugin.jar | relaxngDatatype.jar |
| | soapprocessor.jar | activation.jar |
| ***Katalog: saaj*** | xmlsec.jar | mail.jar |
| saaj-api.jar | xws-security.jar | ***Katalog: jaxp\lib\endorsed*** |
| saaj-impl.jar | xws-security_jaxrpc.jar | xalan.jar |
| | | xercesImpl.jar |
| | | dom.jar |

Tabell 8.1: Nödvändiga jar-filer för att kunna exekvera en klient.