Analys

Syftet är att omvandla kravspecifikationerna för denna iterations funktionalitet till en mer mjukvarulik form (klasser, delsystem osv). Analysen fokuserar på de funktionella kraven, dvs på en ideal bild av systemet där den krassa verkligheten inte gör sig påmind. Detta innebär att indata till analysen främst är use case-modellen. Utdata är klasser, attribut osv men dessa är "logiska", dvs inte nödvändigtvis klasser och attribut i det program vi ska skriva.

Analysens enda syfte är alltså att vara en brygga från kravspec till design för att underlätta designen som är en väsentligt viktigare aktivitet.

När vi nu drar igång modelleringen finns det anledning att än en gång påpeka att artifakter skapas för att vi har nytta av dem. Det finns gott om tips för hur de kan se ut för att vara användbara (tex RUP-mallarna) men det finns ingen lag som förbjuder oss att utforma dem annorlunda om vi vill. Har vi ingen nytta av dem gör vi dem inte alls.

En relaterad fråga är om artifakter ska uppdateras ifall vi vid ett senare skede i modelleringen kommer på att de var felaktiga. Även här gäller att vi bara gör det om det är till nytta, dvs om det är artifakter som kommer att användas igen senare. Vad gäller analysen kanske det är lämpligt att uppdatera alla analysartifakter så länge vi håller på med analys men knappast när vi går över till design.

Exempel

Som exempel tar vi ett påhittat use case "överför pengar mellan konton" för ett banksystem. Aktörer i detta use case är:

- Bankkassören som hanterar överföringen.

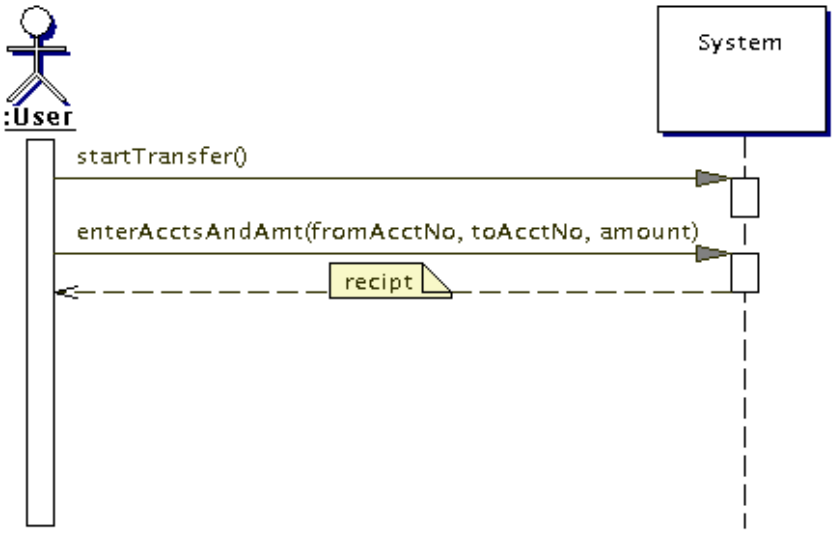
Vi nöjer oss med main success scenario, här kommer det:

1. Kunden kommer till kassan med en blankett för överföring.
2. Kassören påbörjar överföring.
3. Kassören slår in kontonummer (2 st) och belopp.
4. Systemet beräknar eventuella uttagsavgifter
5. Systemet uppdaterar sitt interna data med överföringen.
6. Systemet loggar överföringen.
7. Systemet skriver ut kvitto.
8. Kunden tar kvittot och går sin väg.

System Sequence Diagram

System Sequence Diagram (SSD) är en artifakt som erfaren utvecklare förtrogen med systemet kanske skulle hoppa över. Att utveckla SSD ska inte ens för oss oerfarna stackare ta mer än några minuter.

Syftet med SSD är att tydliggöra texten i scenariot i form av operationer som aktörer utför på systemet, deras ordning, vilka parametrar de har samt hur systemet reagerar på dem. SSD säger inget om vad som händer inuti systemet. Operationerna i SSD är rent logiska, de har inget med metodnamn att göra. SSD är ett sekvensdiagram, här kommer det:



Domain Model (kärt barn har många namn: domänmodell, objektmodell, ...)

Domänmodellen är en logisk bild (symboliserar alltså inte blivande klasser i programmet) av det område i verkligheten som programmet ska hantera (kallas problemdomänen). I vårt exempel är det alltså en bild av banken. I UML är den ett klassdiagram med associationer och attribut men utan operationer. Domänmodellen är också en *abstraktion*, dvs den visar inte hela verkligheten utan bara det som vi tycker är intressant för det blivande programmet. Abstraktioner är ett väldigt bra verktyg. Varje nivå av abstraktion sållar bort något ointressant och visar bara det som är relevant.

Gör vi en bra domänmodell är mycket vunnet när vi går över till design. Det är dock en väsentligt komplexare uppgift än att rita SSD. Här kommer vi för första gången in på den stora vetenskapen om objektorienterad modellering. En något sånär utförlig genomgång av den skulle kräva en hel kurs så det här blir bara några av de viktigaste principerna. För den som vill lära sig mer rekommenderas starkt Larman: *Applying UML and Patterns*.

Steg 1: Klasser

Den klassiska metoden att hitta klasser är att leta efter susbstantiv i scenariot som ska implemeneteras. Här kommer scenariot med substantiven i fetstil:

1. **Kunden** kommer till **kassan** med en **blankett** för **överföring**.
2. **Kassören** påbörjar **överföring**.
3. **Kassören** slår in **kontonummer** (2 st) och **belopp**.
4. Systemet beräknar eventuella **uttagsavgifter**.
5. Systemet uppdaterar sitt interna **data** med **överföringen**.
6. Systemet loggar **överföringen**.
7. Systemet skriver ut **kvitto**.
8. **Kunden** tar **kvittot** och går sin väg.

En annan vedertagen metod är att använda en lista över olika typer av klasser som kan finnas. Nedan följer en sådan lista. För att få fler exempel på klasser finns det också en kolumn för möjliga klasser i ett bokningssystem för flygbiljetter.

Kategori	**Banksystem**	**Flygbokningssystem**
Fysiska objekt som går att ta på	Blankett Pengar	Flygplan
Specifikationer och Beskrivningar	KontoNummer KontorsID (vid vilket kontor	FlightBeskrivning

	kunden är)	
Platser	Bankkontor KundAdress	Flygplats
Transaktioner	Överföring	Reservation
Rader i transaktioner	ÖverföringsRad (datum, kontonr, belopp osv)	
Roller	Kund, Kassör	Pilot
Behållare för saker		Flygplan
Saker i behållare		Passagerare
Externa system		
Abstrakta substantiv		Flygskräck, Rökare
Organisationer	Bank	Flygbolag
Händelser	Avbrott (i kommunikationen mellan terminalen och systemet)	Landning Krasch Flight
Processer	Överföring	Bokning
Regler	Uttagsavgifter Räntor	Avbokningsvillkor
Kataloger	Kontokatalog (innehåller specifikationer för olika sorts konton)	Tidtabell
Journaler, loggar	Kvitto	
Finansiella värden och tjänster	Belopp	Tillgodohavande
Manualer, böcker och andra dokument		RabattLista

Ett par kommentarer innan vi samanställer domänmodellen:

- Det är vanligare att hitta för få klasser än för många. Var inte rädd för att ta med många klasser på det här stadiet.
- Undvik att tänka att en potentiell klass bara är ett attribut i en annan klass. Undantaget är om det handlar om något vi "i verkligheten" betraktar som ett tal eller en text. Enligt denna regel blir tex belopp inte en klass medans Överföringsinformation blir det (istället för att vara ett attribut till överföring).
- Många domänmodeller (som tex den här) innehåller någon form av kvitto. Trots det ska det normalt inte finnas någon kvittoklass i domänmodellen eftersom informationen på kvittot kan sammanställas från andra klasser. Undantaget är om kvittot fyller en speciell funktion, tex att ge rätt att lämna tillbaks en vara. För banksystemet ger kvittot i och för sig rätt att ha det insatta beloppet på kontot men eftersom vi i den här iterationen inte har med hantering av felaktiga saldon utelämnar vi klassen Kvitto för närvarande.
- Det är vanligt att glömma klasser som symboliserar specifikationer, tex en klass i en affär som innehåller information om varor (namn, produktnummer, pris, beskrivning). Ta men sådana klasser om det måste finnas information om en tjänst eller ett föremål oavsett om det finns något fysiskt exemplar av det. Som exempel skulle vi i flygbokningssystemet ovan införa en klass FlightBeskrivning som innehåller flightens nummer. Detta nummer kommer då inte att finnas i klassen Flight. Den kan däremot innehålla tex datum eftersom det inte är gemensamt för alla flighter med samma nummer. I vårt exempel kan vi ha en klass KontoBeskrivning som innehåller räntenivån men inte kontonummer.

Med hjälp av både substantiven och kategorilistan inför vi dessa klasser i domänmodellen:

Observera till slut att det finns bättre och sämre modeller, men däremot inte riktiga och felaktiga.

Steg 2: Associationer

Att hitta associationer är knappast lika viktigt som att hitta klasser men de är ett bra sätt att gör domänmodellen lättare att förstå. Kom ihåg att det fortfarande inte har någontinf med associationer i programmet att göra, de är bara symboler för associationer i "verkliga livet".

Vi kan dela upp associationerna i två olika typer:

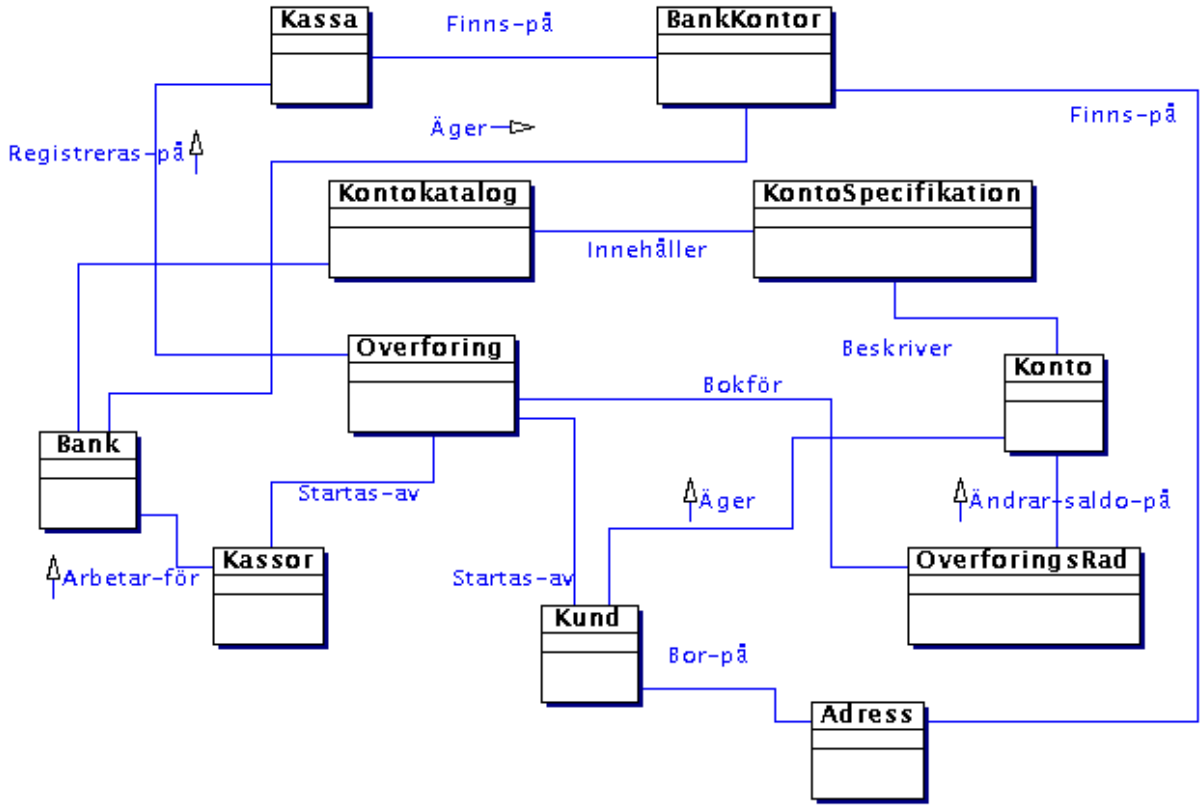
1. **Måste-känna-till**
   Konto måste känna till KontoSpecifikation annars är det omöjligt att veta vilken ränta kontot har. Dessa associationer måste vara med i domänmodellen.
2. **Förståelse**
   Det är inte nödvändigt att Överföring känner till Kund, men sådana associationer kan tas med ändå om de ökar förståelsen av domänmodellen. Till förståelse-kategorin räknas också associationer som kan härledas ur andra associationer: Om A känner till B och B känner till C behövs ingen association mellan A och C. Ta inte med så många förståelse-associationer att modellen blir grötig.

Vi kan hitta associationer antingen genom att kolla på domänmodellen och klura eller genom att använda en sådan här lista över möjliga associationer:

Kategori	**Banksystem**
A är en fysisk del av B	Kassa - Kassaskrin
A är en logisk del av B	ÖverföringsRad - Överföring
A finns fysiskt inuti B	Kassa - BankKontor
A finns logiskt inuti B	KontoKatalog - KontoSpecifikation
A är en beskrivning av B	KontoSpecifikation - Konto
A är en rad i en transaktion eller en rapport av typ B	ÖverföringsRad - Överföring
A loggas i B	Överföring - Kvitto
A är medlem i B	Kassör - Bank
A är organisatoriskt en avdelning i B	BankKontor - Bank
A använder B	Kassör - Kassa Kund - Blankett
A kommunicerar med B	Kund - Kassör
A är relaterad till en transaktion av typ B	Kund - Överföring Kassör - Överföring
A och B är transaktioner med	

inbördes relation	
A är i positionen efter B i någon form av datasamling	ÖverföringsRad - ÖverföringsRad
A ägs av B	Bank - BankKontor Kund - Konto

Det kan underlätta försåelsen att lägga till namn på associationer. Det är bra att använda verb för att namnge associationerna och att försöka få en förståelig mening av klassnamn, associationsnamn, klassnamn. Så här blir domänmodellen med associationer:



Steg 3: Attribut

Attribut symboliserar information som en viss klass måste komma ihåg. Kravspecen (tex use case-modellen) är en källa för att upptäcka sådan information.

Lämpliga kandidater till attribut

Kandidater till attribut är allt data där det inte är menigsfullt att skilja på olika instanser med samma värde. Det är till exempel ingen skillnad på två olika tal som båda har värdet 5, men det kan mycket väl vara skillnad på två olika personer som heter Marie Persson. Tal är alltså lämpligt som attribut medans person passar bättre som en klass. Praktexempel på bra attribut är sådant som vi "i verkligheten" betraktar som siffror, text, klockslag, datum eller boolska värden. Andra bra kandidater är adress, färg, telefonnummer, personnummer, postnummer, uppräkningsbara typer, varunummer, position och storlek.

Dåliga attribut (i domänmodellen) är sådana som ersätter associationer mellan klasser. I klassdiagrammet ovan vore det inte alls bra att ge klassen Kund ett attribut av typen Konto för att visa att kunden har ett konto. Detta ska i stället visas med en association mellan Kund och Konto (så ser det också ut i diagrammet).

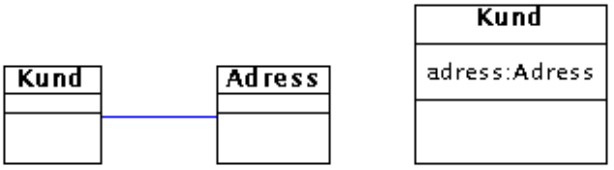
Vissa abstraktioner som uppfyller kriteriet att det inte är menigsfullt att skilja på olika instanser med samma värde gör sig ändå bäst som attribut. Detta gäller till exempel om den

- består av olika delar. En adress består av gata, gatunummer, postnummer, c/o, ...
- har operationer, vanligtvis parsning eller validering. Ett personnummer måste valideras för att kolla att kontrollsiffran stämmer.

- i sin tur har attribut. En räntenivå kan ha ett startdatum och ett slutdatum.
- är en storhet med en enhet. Belopp har en enhet, tex SEK.

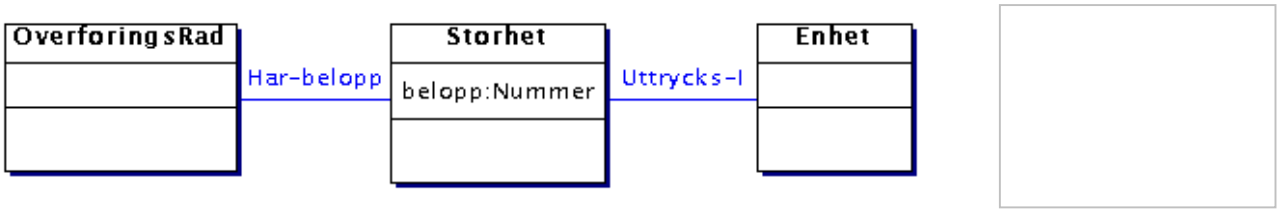
Om det är tveksamt ifall en viss abstraktion gör sig bäst som klass eller attribut är det lämpligt att låta den vara en klass.

Primitiva datatyper (nummer, text osv) som ändå representeras som klasser därför att de faller under någon av punkterna i listan ovan kan vara OK att representera enligt bilden till höger nedan om det är klart att det verkligen handlar om en klass (Adress i detta fall). Övriga klasser ska alltid visas enligt bilden till vänster, dvs med en association och inte som ett attribut.



En kommentar om UML

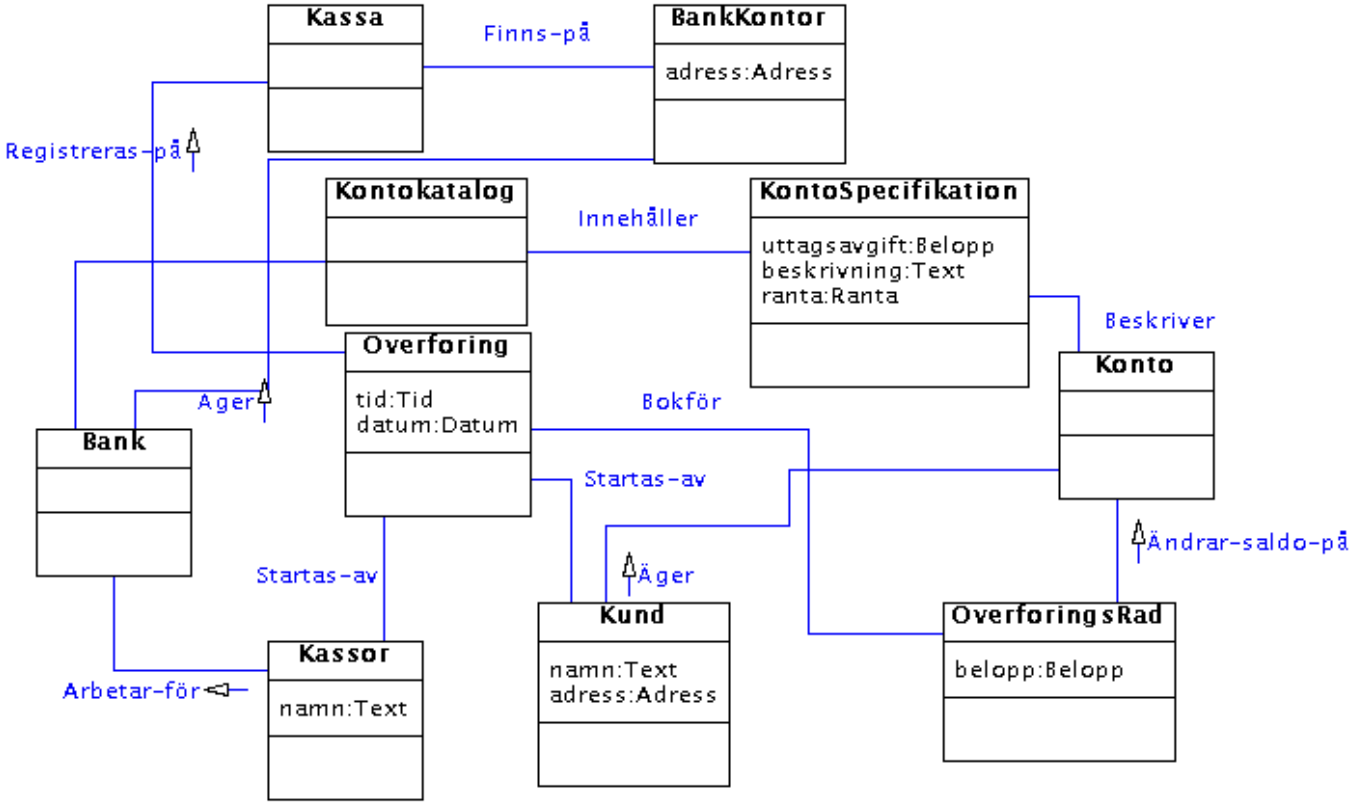
Vi vill visa att ÖverföringsRad har ett belopp. Eftersom belopp har en enhet vill vi att det ska vara en klass. Detta kan visas på flera olika sätt, tex dessa två:



Båda sätten helt OK. Det är dock inte OK att rita som nedan eftersom det säger att det inte finns någon klass belopp.

Attribut i bankexemplet

Så är det dags för den sluliga domänmodellen:


Kassa
Finns-på
BankKontor
adress:Adress
Registreras-på
Kontokatalog
Innehåller
KontoSpecifikation
uttagsavgift:Belopp
beskrivning:Text
ranta:Ranta
Beskriver
Konto
Overforing
tid:Tid
datum:Datum
Ager
Bokför
Bank
Startas-av
Ändrar-saldo-på
Äger
Startas-av
Kund
namn:Text
adress:Adress
OverforingsRad
belopp:Belopp
Arbetar-för
Kassor
namn:Text