Objektorienterad programutveckling i ett nötskal

Tommy Olsson, Institutionen för datavetenskap, Linköpings universitet, © 2014

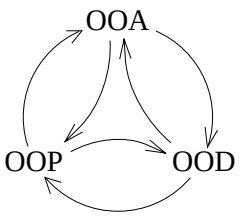
Detta häfte ger en översikt av grundläggande begrepp inom objektorienterad programutveckling: CRC-kort och UML (Unified Modeling Language) för att dokumentera analys och design och en enkel metod för att genomföra objektorienterad analys och design. Det finns mycket osagt i de olika sammanhangen och för mer ingående beskrivningar hänsvisas till litteraturlistan.

Utgångspunkten för all systemutveckling är en kravspecifikation, vilken anger vad systemet ska klara av. Innan ett system kan konstrueras analyseras kravspecifikationen. Syftet med analysen är att erhålla en modell av systemet i form av en mer exakt och mer fullständig specifikation och därigenom en bättre förståelse av systemet och dess relation till omgivningen. I objektorienterad programutveckling görs en *objektorienterad analys* (OOA). Det innebär exempelvis att ta fram *användningsfall* (use cases) med tillhörande figurer och beskrivningar, liksom olika diagram, som *klassdiagram och interaktionsdiagram*. Annan dokumentation som tas fram kan gälla prestandakrav, maskinvara, användning av befintlig programvara, anpassning till standarder, etc.

De diagram och beskrivningar som tas fram i analysfasen utgör en grund för nästa fas, *objektorienterad design* (*OOD*). En fördel med den objektorienterade modellen är att diagrammen från analysfasen kan användas även i designfasen. Under designfasen förfinas och modifieras klasserna som analysfasen givit upphov till och det tillkommer normalt ytterligare klasser vars behov man inser först under designarbetet. Resultatet av designfasen är detaljerade beskrivningar av systemet, dess gränssnitt mot omvärlden, klasser och objekt, hur data ska lagras permanent, etc. Efter designfasen kodas systemet med objektorienterade programmeringsmetoder.

Gränsen mellan *OOA* och *OOD* är inte skarp, även om de två faserna är inriktade på distinkta aktiviteter. I analysfasen försöker man *modellera* systemet genom att upptäcka klasser och objekt som tillhör problemområdet. I designfasen *konstruerar* man abstraktioner som ger det beteende som modellen kräver.

Objektorienterad systemutveckling är en iterativ process där man vid behov går tillbaka och gör om tidigare steg. Det är också troligt att man under arbetets gång har behov av att koda prototyper eller implementera klasser för att prova en viss konstruktion eller ett gränssnitt. Detta arbetssätt kan till åskådliggöras med den så kallade *basebollmodellen*, se figuren till höger.



Verktyg

För att dokumentera hur långt i arbetet man kommit, liksom slutresultatet av varje aktivitet i analys och designfaserna, finns verktyg som stöd, till exempel datorbaserade grafiska analys- och designverktyg. Enkla manuella metoder kan också fungera bra i vissa fall och är dessutom lätta att använda. Post-it-lappar och en skrivtavla kan vara mycket praktiskt att arbeta med i inledande faser.

Nedan ges en översikt av olika hjälpmedel som kan användas för att dokumentera analys- och designarbetet.

Användningsfall

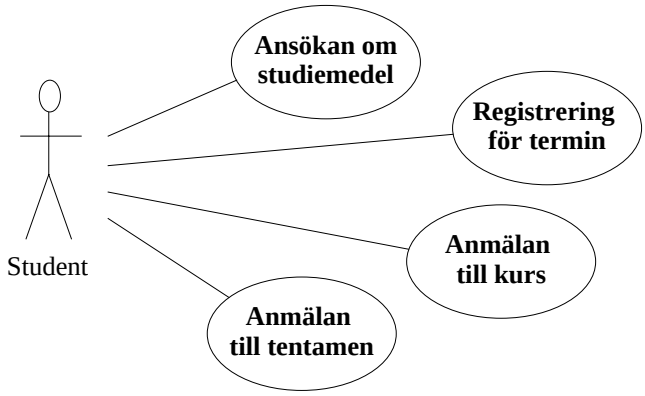
Ett *användningsfall* (use case) är en interaktion mellan en användare och systemet. Om man arbetar med dokumenthanteringsprogram kan ett användningsfall vara att ”ändra ett textstycke till kursiv stil” eller ”skapa en innehållsförteckning”. Ett användningsfall

- utgör en funktion som är synlig för användaren
- uppnår ett distinkt mål för användaren
- kan vara stort eller litet

I enkla fall kan man hitta användningsfall genom att tala med användare och diskutera vad de vill kunna göra med systemet. Varje distinkt sak en användare vill göra ges ett namn. Man skriver också en kort textuell beskrivning på några få meningar. Med en sådan kortfattad beskrivning av ett användningsfall som utgångspunkt utvecklar man sedan iterativt användningsfallet i detalj. Resultatet av en sådan *användningsfallsmodellering* är ett antal aktörer och en katalog med användningfall.

En *aktör* är en roll som en användare har i förhållande till systemet. Aktörer är externa i förhållande till systemet och utbyter information med systemet. En aktör kan utföra olika användningsfall och ett användningsfall kan utföras av flera olika aktörer. Ibland kan det vara givande att först hitta aktörerna i ett system och utifrån dessa komma fram till olika användningsfall.

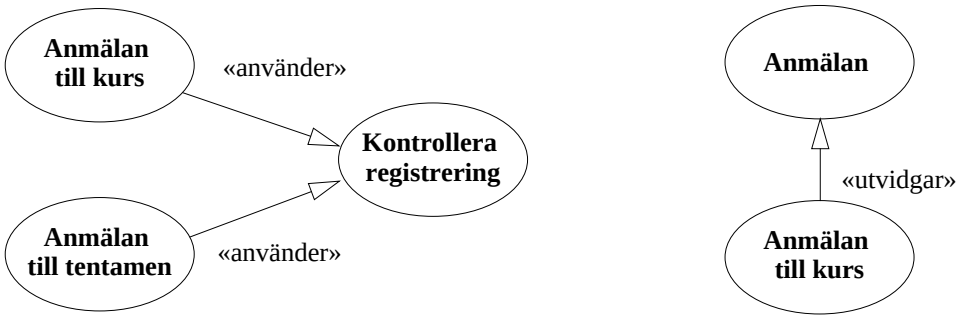
Ett användningsfall utgör en viss funktionalitet i systemet som en aktör använder. Det initieras av aktören och består av en sekvens av händelser i systemet. Användningsfall visualiseras i form av *användningsfallsdiagram* (use case diagram), se figur 1 nedan.



Figur 1. Fyra användningsfall för aktören Student.

Aktörer visas som streckgubbar och användningsfall som ellipser med användningsfallets namn inskrivet. Aktören länkas ihop med de användningsfall som aktören utför med linjer. Förutom de länkar mellan aktör och användningsfall som visas i figur 1, vilka representerar relationen *utför* (carry out), kan det förekomma ytterligare två slags länkar som representerar relationer mellan användningsfall, nämligen *använder* (uses) och *utvidgar* (extends).

Relationen *använder* används då det finns gemensamma delar i olika användningsfall. För att undvika att kopiera beskrivningen för en sådan gemensamt del gör man ett användningsfall av den och inför relationen *använder* i diagrammet mellan de berörda fallen och denna gemensamma del. Relationen *utvidgar* används då man har ett användningsfall som är likartat med ett annat men där det ena gör lite mer. I figur 2 visas hur dessa två relationer anges med pilformade förbindelser och speciella etiketter.



Figur 2. Notation för relationerna använder och utvidgar.

Utförandet av varje användningsfall beskrivs i text. I datorbaserade grafiska verktyg brukar man kunna öppna ett textfönster för varje användningsfall, där texten kan skrivas för att sedan kunna döljas då man inte har behov av att se den.

Ett användningsfall kan ofta utföras på lite olika sätt, till exempel att den ordning i vilken ingående delmoment utförs kan varieras något eller att ett delmoment kan utföras eller ej. Det kan alltså finnas olika vägar att ta sig genom ett användningsfall. En beskrivning av en viss väg att ta sig igenom ett användningsfall, en instans av användningsfallet, kallas för *scenario*. Scenarier kan beskrivas i *interaktionsdiagram*, av vilka det finns två former, *sekvensdiagram* (sequence diagram) och *samarbetsdiagram* (collaboration diagram). Interaktionsdiagram används mest i designfasen och beskrivs längre fram.

Användningsfall använda i ett tidigt skede av processen vara ett sätt att hitta objekt.

CRC-kort

Klasskort är ett enkelt verktyg för att dokumentera klasser och används som ett alternativ eller komplement till diagram. De kan flyttas omkring, grupperas och ordnas på olika sätt som överskådligt visar de olika klasserna och deras samband. Den mest kända typen av klasskort är *CRC-kort*. De är ej en del av UML men kan med fördel användas ihop med UML.

CRC-kortet tog ursprungligen fram i syftet att lära ut objektorienterad programmering men det har blivit en standard. För varje klass används ett CRC-kort. CRC står för *Class, Responsibilities* och *Collaborators*, vilket kan översättas med *klass*, *ansvar* och *samarbetspartners*. Klass avser att man ska skriva klassens namn på kortet. Ansvar avser att man kortfattat skriver ner de viktigaste ansvaren/syftena som klassen har. För varje ansvar anger man vilka andra klasser, samarbetspartner, som klassen måste samarbeta med för att utföra uppgiften.

CRC-kort ser i princip ut som i figur 3. Överst finns en rad där man skriver klassens namn. Resten av kortet är delat vertikalt. Till vänster anges klassens ansvar och för varje sådant ansvar anger man till höger vilka andra klasser som samarbete förekommer med.

Beställning	
Kontrollera om vara finns i lager	Beställningsrad, Lager
Bestäm pris	Beställningsrad, Prislista
Kontrollera betalning	Kund
Avsänd till leveransadress	Kund

Figur 3. Ett CRC-kort

Kortet i figur 3 avser att beskriva klassen **Beställning**. En beställning förutsätts här bestå av ett antal rader på en beställningsblankett. På varje **Beställningsrad** anges en produkt, dess á-pris och det antal som beställs. På beställningen anges också vilken **Kund** som beställt produkterna. Andra klasser som samarbete sker med är **Lager** och **Prislista**, för att kontrollera om produkterna finns i lager och om priset på beställningen överensstämmer med aktuell prislista. Ett ”riktigt” CRC-kort har storleken 4x6 tum, dvs ungefär som ett registerkort. Det finns en medveten avsikt med att inte använda större kort; – det är inte tillåtet skriva mer än vad som ryms på kortet.

En av de stora fördelarna med CRC-kort är att de uppmuntrar till diskussion mellan utvecklarna, till exempel då man går igenom ett användningsfall för att se hur det kommer att implementeras av klasserna. Utvecklarna plockar fram korten allteftersom varje klass medverkar i ett användningsfall. Successivt som varje klass’ ansvar framkommer kan man skriva ner det på CRC-kortet. Det är viktigt att fundera på det ansvar en klass har, eftersom det leder bort från synsättet att klasserna är passiva databehållare och i stället underlättar förståelsen av deras beteende i stort.

Ett vanligt misstag är att man gör långa listor med ansvar beskrivet för detaljerat. Det bör helst inte vara fler än 3-4 viktiga ansvarspunkter. Kommer man fram till flera kan man överväga om klassen bör delas upp i flera klasser eller om ansvaret ska beskrivas på en högre nivå. För att samla och formalisera resultatet av CRC-modelleringen i en UML-beskriven design kan man använda klassdiagram och interaktionsdiagram.

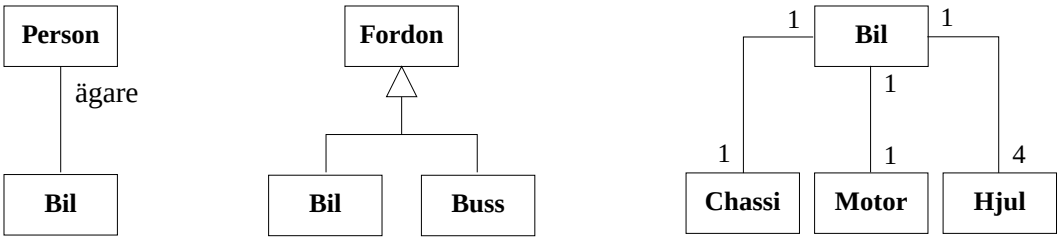
Klassdiagram

Ett klassdiagram beskriver olika typer av objekt i ett system och statiska relationer mellan dessa. I princip kan man särskilja två slags relationer, *association* och *subtypning* (härledning/arv):

- en *association* representerar någon form av relation mellan instanser av klasser, till exempel att en person kan äga en bil eller att en bil har en motor.
- *subtyp* innebär att en typ är en specialisering av en annan typ, till exempel att en bil är ett *specialiserat* fordon, vilket också exempelvis en buss är. Man kan också se det som att fordon är en *generalisering* av

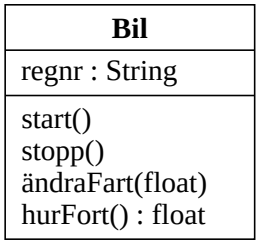
alla fortskaffningsmedel som betecknas fordon. En sådan *supertyp*, *superklass*, eller *basklass*, beskriver subtypernas, *subklassernas*, gemensamma egenskaper.

En klass ritas, i det enkla fallet, som en rektangel med klassens namn skrivet i rektangeln. En association ritas som en enkel linje mellan relaterade klasser. Subtypning, eller *arv*, ritas med en triangelformad symbol (discriminator) vid basklassen och en enkel linje till subklassen. Om det finns flera subklasser till en klass (basklass) kan man låta linjen förgrena sig till varje subklass. Linjerna mellan klasserna kan kompletteras med information av olika slag, till exempel multiplicitet (en bil har *4* hjul) eller rollnamn (en person är *ägare* till en bil). I figur 4 nedan visas tre enkla klassdiagram.



Figur 4. Några enkla klassdiagram.

Diagrammet till vänster i figur 4 visar att en Person kan äga en Bil. Detta är en association, vilken förtydligats genom att *rollnamnet* ägare anges vid Person. I mitten visas en *arvshierarki*, nämligen att Fordon är en generalisering av Bil och Buss (eller att Bil och Buss är specialiceringar av Fordon). Diagrammet till höger visar att en Bil består av ett Chassi, en Motor och Hjul. Dessa relationer är associationer och i diagrammet har *multipliciteter* för dessa angivits; *en* Bil har *ett* Chassi, *en* Motor och *fyra* Hjul.



Figur 5.

Rollnamn och multiplicitet är bara två exempel på hur diagram kan göras mer detaljerade. Klasser kan beskrivas mer detaljerat genom att dela upp rektangeln i tre delar, se figur 5. Den första delen används för *klassnamnet*, den andra delen för klassens/objektens *attribut* och den tredje delen för *operationer* som kan utföras på klassen/objekten. För attribut kan man ange datatyp och initialvärde, om så önskas, och för operationer kan man ange argumentlista och returvärdestyp.

Association, ibland kallad *känner-till*-relation, kan användas för att beskriva alla former av relationer som ej är subtypning. I UML används ytterligare några former av relationer, *aggregation* och *composition*. Aggregering kallas också *har*-relation eller *del-av*-relation. Relationen är komplicerad, vilket framgår av att olika auktoriteter inom området inte är överens om vad *del-av* egentligen är i alla sammanhang. Vissa anser exempelvis att relationen mellan ett företag och dess anställda är en aggregering, medan andra anser att det snarare är en association (en lösare relation). Man får konstatera att det inte alltid finns en klar distinktion mellan association och aggregering. Komposition är en starkare form av aggregering, där ett delobjekt endast kan ingå i *en* helhet och att delobjektet normalt endast existerar medan denna helhet existerar.

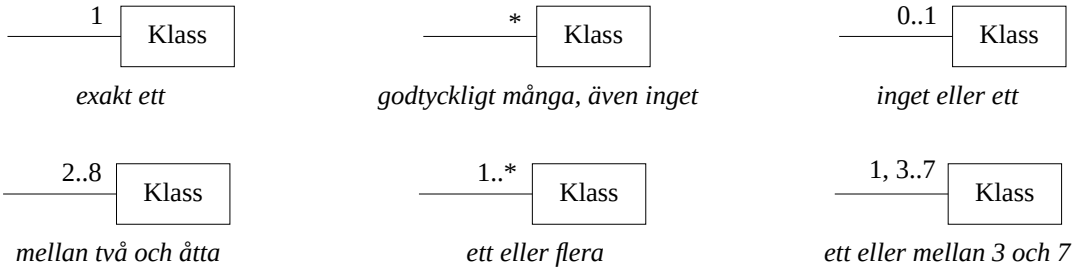
I figur 4 ovan, t.h., finns ett klassdiagram som beskriver en bils beståndsdelar. De relationer som förekommer är *del-av*-relationer. Man kan anse att chassit är starkt förknippat med bilens existens; då bilen skrotas försvinner chassit. Motor och hjul, däremot, kan bytas ut. Diagrammet kan utifrån detta modifieras enligt diagrammet till vänster i figur 6.

Figur 6. Aggregering och komposition.

Relationen mellan Bil och Chassi har ersatts med relationen komposition, vilket i UML betecknas med en fylld romb vid det sammansatta objektets klass. Relationerna mellan Bil och Motor resp. Hjul har ersatts med aggregering, vilket i UML betecknas med en ofylld romb.

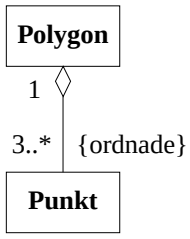
Till höger i figur 6 visas även ett alternativt sätt att rita komposition. Komponenten ritas i detta fall inuti helheten. Dess klassnamn skrivs med normal stil (ej fet) och på formen *rollnamn*:*Klassnamn*. Ettan uppe till höger anger multipliciteten.

Multiplicitet har använts i några diagram ovan. Det finns några grundläggande former som används för att beteckna ett obligatoriskt antal, ett godtyckligt antal eller ett valbart antal. Några exempel på olika möjligheter att ange multiplicitet visas i figur 7 nedan.



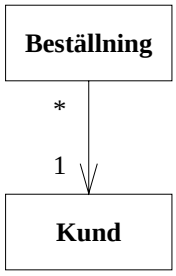
Figur 7. Exempel på multipliciteter.

Mycket av det som ritas i diagram anger *begränsningar* (constraints). I vissa situationer finns det behov av att tillföra beskrivningar av dessa. UML har ingen strikt syntax för detta, mer än att sådant beskrivningar ska skrivas inom spetsparenteser, {…}. För övrigt kan man välja mer eller mindre formella sätt att beskriva begränsningarna. Till exempel kan en polygon beskrivas som en ordnad mängd bestående av mist tre punkter (där punkterna i ordning sammanbinds med linjer). Figur 8 visas hur detta kan visas i ett klassdiagram. Punkterna i en Polygon kan ändras då man ändrar Polygonen, därav relationen aggregering. Multiplicititeterna anger att en Polygon består av tre eller flera Punkter, där punkterna är ordnade inbördes.



Figur 8.

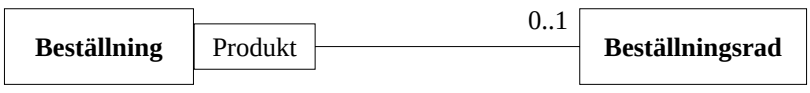
En association kan vara riktad, vilket kallas *navigerbarhet* (navigability). Riktning anger på vilken sida av associationen som ansvaret finns. I en beställning av en produkt anges vilken kund som gjort en beställning. Man kan tänka sig att en beställning har ansvar för att upplysa om vilken kund som gjort beställningen. Detta kan visas i ett diagram genom att rita en pil på strecket mellan Beställning och Kund, se figur 9.



Figur 9.

Ett beroende (*dependency*) föreligger mellan två klasser om ändringar i definitionen av den ena klassen medför förändringar i den andra. Sådana beroenden kan finnas av olika skäl: en klass sänder meddelanden till en annan klass; en klass har en annan klass bland sina attribut; en klass anger en annan klass som parameter i en operation. Beroenden är något man vill undvika och minimering av beroenden ingår som en del i systemdesignen. Det går dock inte alltid att undvika beroenden och det kan vara värdefullt att kunna visa i diagram, vilket görs med en streckad pil.

Antag att ett företag hanterar beställningar från sina kunder. Varje beställning omfattar ett antal rader, där kunden på varje rad anger en produkt och det antal som beställs. Det finns en begränsning i form av att en viss produkt endast får anges en gång i varje beställning. För att ange sådana begränsningar i UML använder man en *begränsad association* (qualified association), se figur 10.

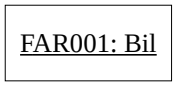


Figur 10. Begränsad association: för varje produkt får finnas endast en beställningsrad.

Interaktionsdiagram

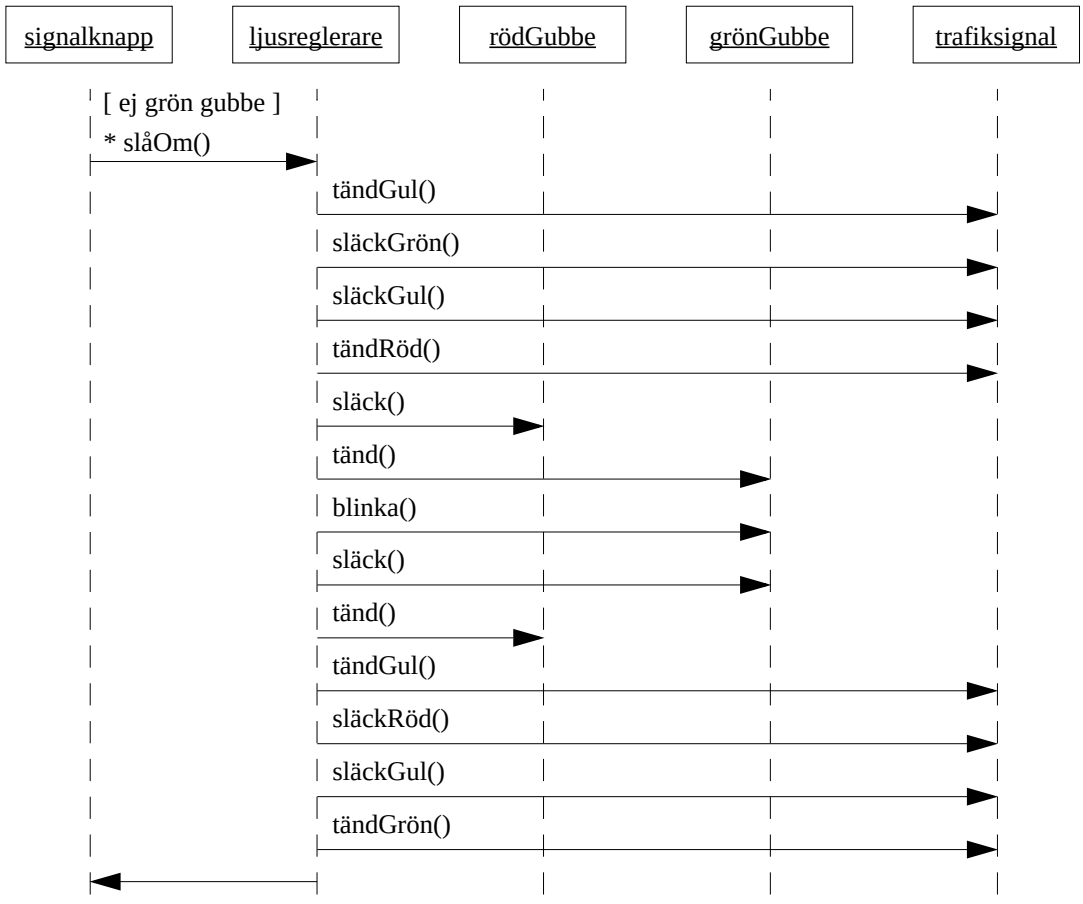
För att visa hur objekt kan samarbeta i ett användningsfall används *interaktionsdiagram* (interaction diagrams). Det finns två slags interaktionsdiagram, *sekvensdiagram* (sequence diagram) och *samarbetsdiagram* (collaboration diagram). Ett sekvensdiagram visar i detalj i vilken tidsordning objekt interagerar, medan ett samarbetsdiagram främst ger en övergripande bild av vilka objekt som samarbetar med varandra.

I interaktionsdiagram förekommer objekt, vilka ritas som rektanglar. I rektanglarna skriver man objektets namn tillsammans med namnet på klassen, på formen *objektnamn*: *Klassnamn*, alternativt enbart objektnamnet. Understruken stil används. Ett bilobjekt som representerar bilen med registreringsnumret FAR001 kan ritas enligt figur 11. I vissa fall är man inte intresserad av att namnge ett specifikt objekt, utan ett objekt i allmänhet av en viss typ och skriver i sådana fall namnet som enBil:Bil eller helt enkelt utan något objektnamn, :Bil.



Figur 11.

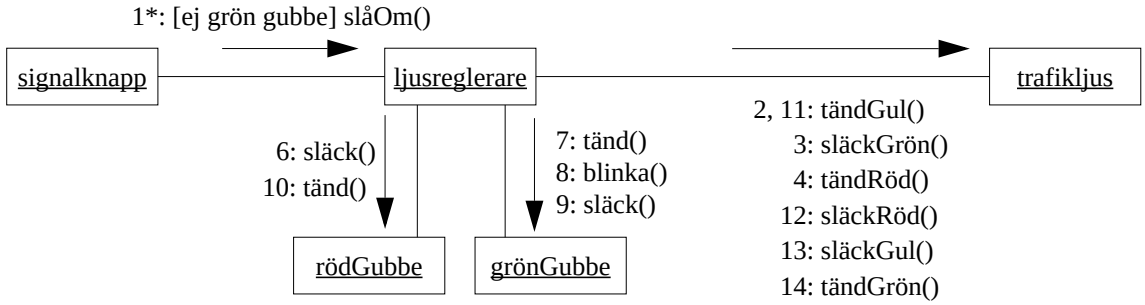
I ett *sekvensdiagram* visas ett objekt som en rektangel ovanför en streckad linje. Den streckade linjen kallas objektets *livslinje* (lifeline) och den visar objektets deltagande i interaktionen. I figur 12 visas ett sekvensdiagram för hur trafikljus och röd/grön gubbe tänds och släcks under en cykel vid ett ljusreglerat övergångsställe, från det att en fotgängare trycket på knappen för att gå över, till dess röd gubbe åter lyser. Trafiksignalen med röd, gul och grön lampa behandlas i diagrammet som ett objekt, medan röd och grön gubbe representeras var för sig.



Figur 12. Ett sekvensdiagram för ett ljusreglerar övergångsställe.

Varje meddelande representeras med en pil mellan livslinjerna för två objekt. Ordningen som meddelanden sänds i visas uppifrån och ner. Varje meddelande märks med åtminstone meddelandets namn. Man kan även ange argument och styrinformation. Det finns ett villkor för att meddelandet slåOm() ska sändas, nämligen att ej grön gubbe är tänd. Asterisken framför meddelandet slåOm() anger att meddelandet kan upprepas (iteration). I diagrammet finns en vänsterriktad pil, som visar en retur från meddelandet slåOm(). Tidsaspekter har utelämnats i detta diagram.

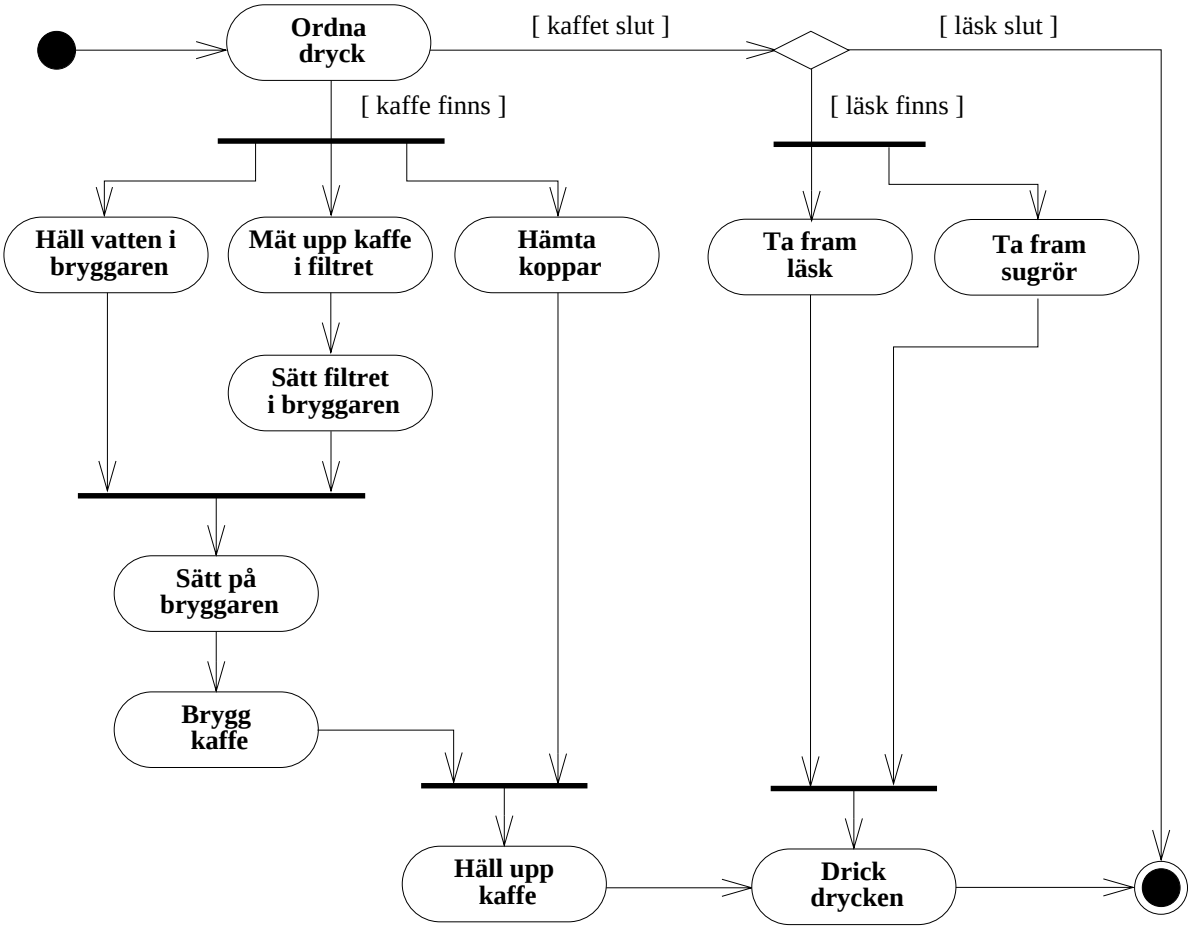
I ett *samarbetsdiagram* (collaboration diagram) visas objekten som ikoner. Som i fallet sekvensdiagram används pilar för att visa meddelanden som sänds, men i detta fall visas sekvensen genom att pilarna numreras. Ett exempel på hur ett samarbetsdiagram kan se ut visas i figur 13 nedan.



Figur 13. Ett samarbetsdiagram.

Aktivitetsdiagram

Aktivitetsdiagram är speciellt användbara för att beskriva en arbetsgång och för att beskriva förlopp med samtidiga händelser. Ett exempel på ett aktivitetsdiagram för att ordna med dryck, i första hand kaffe, till fikapausen visas i figur 14 (denna typ av diagram har påtagliga likheter med Petri-nät).



Figur 14. Aktivitetsdiagram: Ordna dryck.

Diagrammet i figur 14 innebär följande. Då en person ska ordna med dryck och det finns kaffe, gör personen i ordning kaffe. Aktiviteterna att hälla vatten i bryggaren, att mäta upp kaffe i filtret och att hämta koppar kan utföras i godtycklig ordning. Innan bryggaren får sättas på måste vattnet vara hällt i bryggaren och filtret med kaffet ditsatt. Då kaffet är bryggt och kopparna hämtade kan kaffet hällas upp och sedan drickas. Om det inte finns kaffe men däremot läsk, tar personen fram läsk och sugrör, i någon ordning, varefter läsken kan drickas. Om både kaffe och läsk är slut, så avslutas aktiviteten utan att det bjuds på något.

Ett aktivitetsdiagram har en *ingång* som markeras med en stor punkt och en *utgång* som markeras med en punkt med en cirkel omkring. Den centrala symbolen i ett aktivitetsdiagram är *aktivitet* (activity), vilken ritas som en rundad rektangel med aktivitetens namn inskrivet med fet stil. Det kan också finnas *beslutsaktiviteter* (decision activities) som motsvarar ett val. Sådana ritas som en romb med två utgående avfyrare.

Varje aktivitet kan ha en eller flera *avfyrare* (triggers), som då aktiviteten ifråga har utförts, avfyrar en eller flera direkt efterföljande aktiviteter. En avfyrare ritas som en linje med en pilsymbol pekande på den efterföljande aktiviteten. Varje avfyrare har en *gard* (guard), som är ett logiskt uttryck, som beräknas till sant eller falskt och avgör om avfyrning ska ske. Garder skrivs inom klamrar.

I aktivitetsdiagrammet kan vidare finnas *synkroniseringslinjer* (synchronization bars), från vilka flera avfyrare kan utgå. Synkroniseringslinjer används för att ange att efterföljande aktiviteter kan utföras samtidigt, sammanflätat eller helt enkelt att den ordning aktiviteterna utförs i är oviktig.

Analys- och designmetodik

I mindre sammanhang, till exempel undervisning, där man ska lösa mindre uppgifter eller göra små projekt, kan följande grundläggande analys- och designaktiviteter vara tillräckligt:

- Finn objekt och klasser
- Finn relationer mellan objekt och klasser
- Identifiera och beskriv användningsfall
- Utför olika scenarier för användningsfallen, för att gå igenom händelseförlopp i detalj och kontrollera att inget saknas i modellen
- Finn operationer och attribut

Med de klassdiagram, klassbeskrivningar, användningsfall och scenarier som detta ger kan man utföra kodning av små program. Vad dessa punkter mer i detalj omfattar framgår nedan, där en lite mer omfattande metod beskrivs.

Objektorienterad analys (OOA)

Analysfasen utgår från en kravspecifikation och målet är huvudsakligen att generera en uppsättning klass- och objektdiagram med tillhörande beskrivande texter. Det finns olika metoder beskrivna i litteraturen för att göra detta. Varje metod bygger på en process som specificerar vilka aktiviteter som ska utföras och hur de ska utföras. Vid behov itererar man och man kan även i vissa situationer tänkas hoppa mellan olika aktiviteter. Följande aktiviteter förekommer då man analyserar ett system eller delsystem:

1. Finn objekten
2. Klassificera objekten
3. Beskriv relationer mellan klasser
4. Gruppera klasser
5. Identifiera och beskriv användningsfall och utför scenarier för att verifiera (del)systemet

1. Finn objekt

Ett förslag till hur man ska finna objekt är att läsa kravspecifikationen (del av) och notera ofta förekommande *substantiv*. Sådana substantiv kan ofta vara kandidater till objekt. (på samma sätt kan associerade *verb* påvisa operationer på objekten).

En annan möjlighet är att projektgruppen genomför en gemensam idékläckning (brainstorming). Utifrån en beskrivning av ett problem försöker man komma på tänkbara objekt. Inga objektkandidater ska förkastas, syftet är enbart att generera en lista av *tänkbara* objekt. Man ska inte heller fundera över hur objekt eventuellt interagerar, det är en senare fråga. Denna metod har varit (är) mycket populär men det har visat sig att det ofta ger bättre resultat om var och en tänker igenom problemet först och att man sedan samlas och diskuterar olika förslag.

Varje förslag till objekt prövas sedan och man stryker sådana som ej anses relevanta. Det kan också visa sig att flera förslag egentligen avser samma objekt och väljer då den beteckning som man finner bäst.

Man kan även söka efter objekt enligt vissa kategorier. Följande är exempel på vanliga sådana objektkategorier (olika författare arbetar med lite varierande kategorier):

- *faktiska ting*. Ex: bil, hus, sensor, givare, kassa.
- *platser*. Ex: kontor, rum, etc.
- *begrepp*; ej fysiska ting men centrala i systemet. Ex: bankkonto, transaktion.
- *roller*. Ex: kund, kassörska.
- *händelser*. landning, avbrott, transaktion.
- *interaktioner*. Ex: lån, möte, byte, etc.
- *yttre enheter* med vilka systemet interagerar.

Finn-objekt-aktiviteten resulterar i en lista över objekt som man efter prövning kommit fram till finns i systemet. Som komplement kan man göra en skiss över objekten, där objekten kan vara preliminärt grupperade efter samhörighet.

2. Klassificera objekt

Alla objekt som finn-objekten-aktiviteten identifierat ska klassificeras. Detta innebär att man ska bestämma klasser som objekten är instanser av. Beskriv varje klass kortfattat med:

- klassens namn – välj namn med omsorg!
- ansvar – operationer som kan utföras på objekt av klassen ifråga.
- samarbetspartners – vilka andra klasser som klassen samarbetar med för att utföra sina åtaganden.

Vilka andra objekt/klasser ett objekt behöver känna till kan upptäckas genom att ställa frågor som ”varifrån får objektet denna information”, ”till vem ska objektet leverera denna information”. Ibland kommer kunskaper att representeras av en variabel i objekten ifråga, ibland i form av utdata från anrop av metoder i andra objekt. CRC-kort kan med fördel användas för att dokumentera, se ovan.

3. Beskriv relationer

I detta steg bestämmer man de relationer som finns mellan klasserna i systemet. Genom detta finner man grupper av klasser som hör ihop, samarbetar eller på annat sätt interagerar. Relationer kan vara statiska eller dynamiska. En relation är antingen arv eller association (som eventuellt kan kategoriseras som aggregation eller composition). Ett eller flera klassdiagram ritas för att dokumentera relationerna.

4. Gruppera klasser

Föregående steg påvisar olika former av samhörighet mellan klasser och objekt. Detta kan vara en utgångspunkt för att dela in klasserna i olika grupper som kan representera till exempel delsystem. Ett annat kriterium kan vara att dela upp i tillämpningsspecifika klasser resp. mer generella infrastrukturklasser. Använder man CRC-kort kan man enkelt gruppera klasser genom att lägga korten i olika högar eller under olika flikar i ett kortregister.

5. Identifiera aktörer och användningsfall

Ett användningsfall är en interaktion som kan inträffa under systemets exekvering. Syftet med att identifiera aktörer och användningsfall och utföra olika scenarier för användningsfall är att få en djupare insikt om samarbetet mellan objekt. Under denna aktivitet kan man till exempel upptäcka nya aktörer och användningsfall, nya objekt/klasser, nya samarbetspartners och nya relationer mellan klasser och objekt.

Sammanfattning av analysfasen

Analysfasen har beskrivits i fem delmoment. Ordningen mellan dessa behöver ej vara strikt. Man skulle till exempel kunna tänka sig att i vissa situationer utföra moment 5 före moment 4, eller att i vissa fall arbeta parallellt med flera moment. Ibland upptäcker man behovet av nya klasser och för sådana klasser att upprepas moment 2 och framåt.

Resultatet av analysfasen är ett dokument med klassdiagram som beskriver relationer mellan klasser. Varje klass beskrivs textuellt. Moduldiagram används för att beskriva delsystem. Interaktionsdiagram används för att beskriva händelseförlopp och hur objekt samarbetar. Användningsfall redovisas. Vidare anges till exempel krav på systemet vad avser prestanda, maskinvara, följande av standarder, återanvändning av annan programvara och liknande förutsättningar. Resursplanering, exempelvis planering av tid och personal, ingår också.

Objektorienterad design (OOD)

Efter analysfasen följer designfasen, med dokumentationen från analysfasen som utgångspunkt. Gränsen mellan analys och design är ej skarp och ibland beskrivs båda faserna under begreppet objektorienterad design. Designfasen består ej av ett antal delmoment som ska genomföras i ordning, utan utgörs av några moment som är relativt självständiga.

1. Systemkonstruktion
2. Infrastrukturkonstruktion
3. Detaljkonstruktion

1. Systemkonstruktion

Systemkonstruktionen är av överordnad karaktär och lägger främst ut riktlinjer för hur det vidare arbetet under konstruktionsfasen ska genomföras. Om parallellism finns i systemet kräver det speciella hänsynstaganden vid konstruktionsarbetet, till exempel om man ska ha verklig samtidighet under exekvering eller enbart simulera samtidighet. Ska ett distribuerat system konstrueras kommer frågor kring hur delsystem ska distribueras. Andra övergripande aspekter kan vara koppling till exempelvis databassystem.

2. Infrastrukturkonstruktion

Infrastruktur gäller till exempel användargränssnitts utformning, hur permanent datalagring ska göras, hur kommunikation mellan olika delar i systemet ska ske, etc. En grafisk användargränssnitt är numera ofta en påtaglig del av många tillämpningsprogram.

Eftersom kravspecifikationer ofta innehåller information som gäller infrastrukturen i ett system, har detta arbetet ofta påbörjats redan i analysfasen. Man kan alltså förvänta sig att ha infrastrukturklasser beskrivna i analysdokumentet. Inget speciellt gäller vid konstruktion av infrastrukturklasser, jämfört med andra slags klasser.

3. Detaljkonstruktion

Detta moment gäller konstruktionen av de enskilda klasserna och nu är man således nere på en relativt detaljerad nivå. Namn på metoder, metodernas parametrar, deras namn och typ, returvärden och attribut (datamedlemmar) ska specificeras noggrant.

Under detta arbete kommer man antagligen att upptäcka att vissa av de befintliga klasserna behöver modifieras, vissa kanske behöver delas upp i flera klasser, medan andra klasser kan elimineras. Helt nya klasser kan tillkomma.

Namngivning av klasser, metoder etc. är en viktig del av detaljkonstruktionen och bör ges stort utrymme. Några regler:

- klassnamn bör vara substantiv i singularis
- metodnamn (funktionsnamn) ska vara verb
- undvik för långa namn, förkortningar, prefix eller suffix till namn
- använd någon lexikalisk standard, till exempel att klassnamn skrivs med inledande stor bokstav, att metoder och attribut skrivs med små bokstäver, att om namn är sammansatta inleds varje delord med stor bokstav.
- använd etablerad terminologi

För övrigt tillämpas vanliga goda programkonstruktionsprinciper, som läsbarhet, enkelhet, svag koppling mellan klasser, strukturering, undvikande av globala data, etc.

Efter detaljkonstruktion kan man tänka sig att åter utföra de scenarion man konstruerat och testat i analysfasen, men nu på en mer detaljerad nivå i syfte att verifiera det man konstruerat.

Sammanfattning av designfasen

Designfasen ska ligga till grund för kodning av systemet. Diagram, scenarion och beskrivningar som erhölls som produkt av analysfasen har under designfasen förfinats, kompletterats och ytterligare verifierats. Klassbeskrivningarna har avslutats och är efter designfasen i princip fullständiga, dvs kan direkt användas för att koda klasserna. Namnen på varje klass' datamedlemmar, metoder och metodernas argument är bestämda och typen för alla sådana attribut är bestämd. Dokumentering av klasser och klassmetoder kan göras med klass- och funktionsbeskrivningsformulär, förutom annan beskrivande text.

Objektorienterad programmering (OOP)

Efter att designfasen genomförts ska systemet kunna kodas i ett objektprogrammeringsspråk (*OOPL*), där det specifika programspråket utnyttjas på bästa sätt för att realisera olika konstruktioner. Objektorienterad programmering innebär att man utnyttjar de speciella konstruktioner som finns i objektprogrammeringsspråk, vilket omfattar klasser, arv och polymorft beteende.

En objektorienterad design kan, under vissa förutsättningar, tänkas realiseras även i ett språk som ej är ett fullfjädrat objektprogrammeringsspråk, men som har annat stöd för programmering med objekt.

Litteratur

Bellin D., Suchman Simone S.: *The CRC Card Book*. Addison-Wesley Longman, 1997.

Fagerström J.: *Objektorienterad analys och design – en andra generationens metod*. Studentlitteratur, 1999.

Fowler M.: *UML DISTILLED, Applying the Standard Object Modeling Language*. Addison-Wesley Longman, 1997.

Quatrani, T.: *Visual Modeling with Rational Rose and UML*. Addison-Wesley Longman, 1998.

Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language Reference Manual*. Addison-Wesley Longman, 1998.