Objektorienterad analys och design med CRC-kort

Marie Nordström och Jürgen Börstler

UMINF 02.19

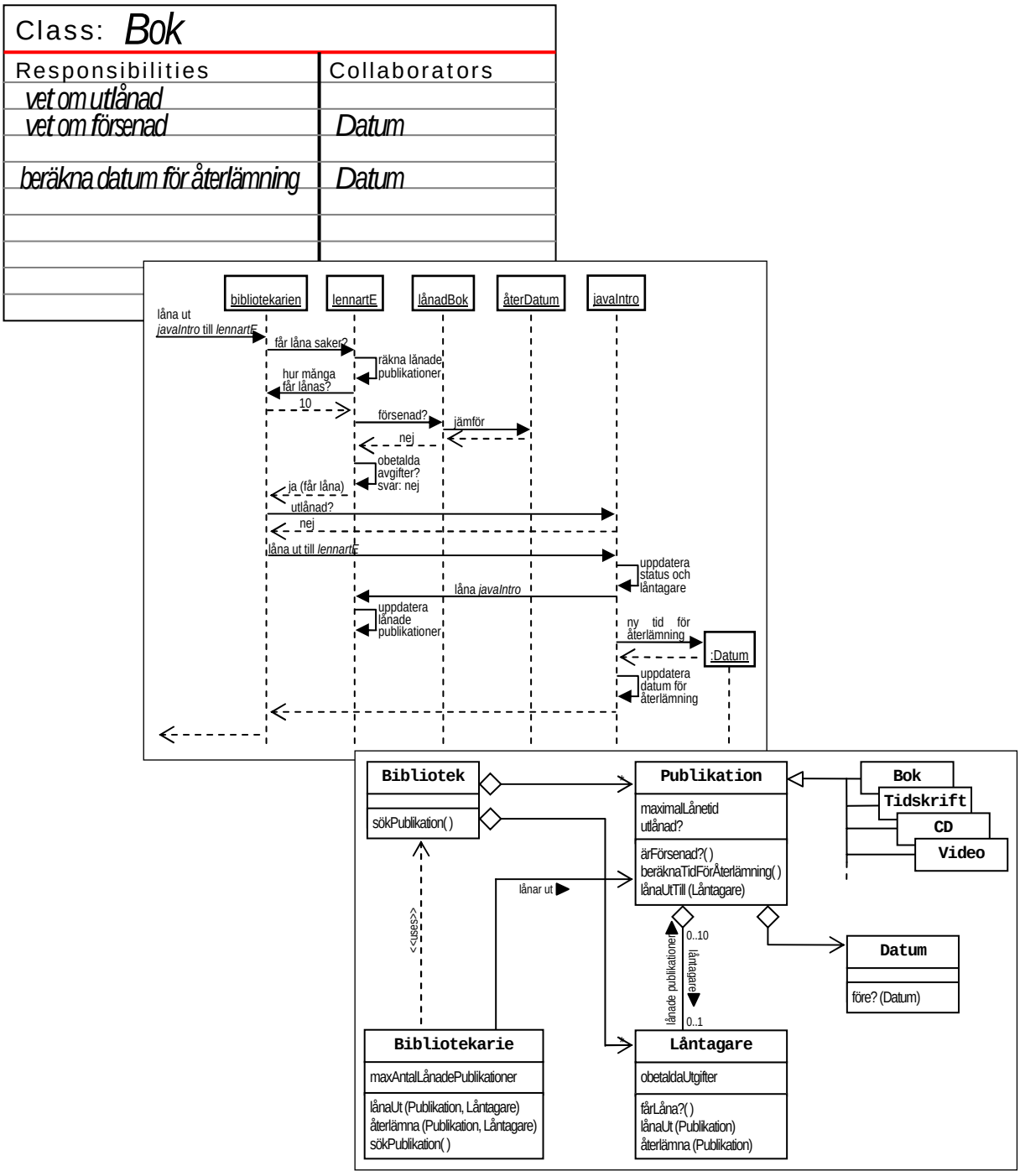
ISSN-0348-0542

(version 3.0, 020924)

UMEÅ UNIVERSITY
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Objektorienterad analys och design med CRC-kort

Marie Nordström, Jürgen Börstler



Innehållsförteckning

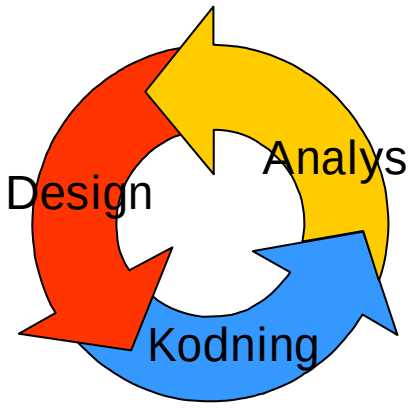
1 Introduktion

Bara för att man jobbar med klasser och objekt behöver det inte betyda att det man åstadkommer är objektorienterat. Vad som behövs är ett tillvägagångssätt (dvs. en metodik) som kan hjälpa till att ta fram problemlösningar som senare kan resultera i välskriven objektorienterad kod som är lätt att förstå, återanvända och underhålla. Det finns inga garantier att en metodik fungerar på alla typer av problem. Ansvaret ligger alltid på att problemlösaren ha ägnat problemet tillräcklig tid och genomarbetning. Objektorienterad analys och design (OOA&D) med CRC-kort, som beskrivs här, har dock visat sig fungera bra i många sammanhang.

Detta är tänkt att vara en introduktion till objektorienterad analys och design och beskriver ett förfarande som innehåller följande delar:

- **Objektorienterad analys (OOA):** målet är att *förstå* problemet och att bygga en objektorienterad modell av problemområdet. Vilka är objekten i detta problem, vilka egenskaper och beteenden har de och hur interagerar de med varandra? Med andra ord; vad *vet* varje objekt, vad *gör* varje objekt och på vilka sätt samarbetar varje objekt med vilka andra objekt. Analysen är oberoende av programmeringsspråk och val av användargränssnittet[1]. Fördelen med att hålla sig ifrån ett specifikt språk och användargränssnitt är att det blir en mer generell lösning. Valet av språk och användargränssnitt kan skjutas på framtiden och man behöver inte bry sig om sådana detaljer i analysen. Dessutom behöver man inte kunna ett specifikt programmeringsspråk för att analysera och förstå ett problem. Det är viktigt att kunna koncentrera sig på det väsentliga i problemet.
- **Objektorienterad design (OOD):** nu är det dags att försöka ta fram en *lösning* på problemet som senare kan implementeras. Då måste detaljer beaktas. Vilka konkreta klasser behövs och vilka faktiska attributvariabler ska de ha för att beskriva vad objekten vet och är? Vilka metoder behövs för att beskriva de beteenden objekten har? Här råder delade meningar om möjligheten att fortsätta att vara språkoberoende. En god rekommendation är nog ändå att försöka begränsa språkberoendet så mycket som möjligt.
- **Objektorienterad programmering (OOP):** till sist skall koden byggas i ett konkret programmeringsspråk.

Det är viktigt att påpeka att detta arbete inte är linjärt! Det krävs att man flera gånger reviderar sin analys och design allteftersom man får större insikt i problemet och därmed förståelse för lösningens möjligheter och begränsningar.



OBS! Det finns ytterligare viktiga aktiviteter som t.ex. testning, dokumentation, etc. som vi dock inte ta hänsyn till här.

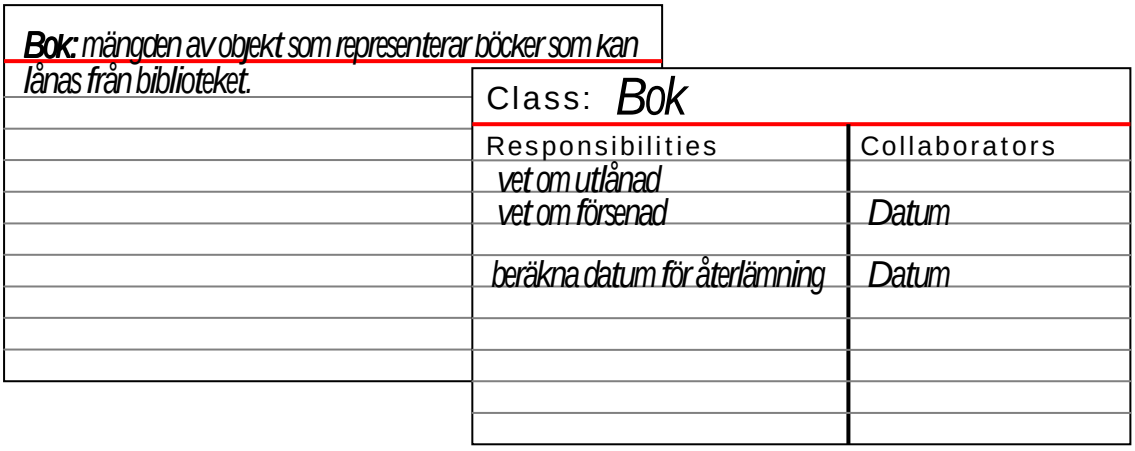
[1] Om inte användargränssnittet i sig är en del av problemet.

2 CRC-kort

CRC-kort är ett hjälpmedel för OO modellering. De används för att lätt kunna ta fram, diskutera och testa olika modelleringsalternativ i OOA&D fasen. Användandet av CRC-kort underlättar att tänka objektorienterad.

CRC står för ***C***lass (klass), ***R***esponsibilities (ansvaren) och ***C***ollaborators (samarbetspartner).

- **Class:** ett CRC-kort motsvarar normalt en klass, dvs. en samling likartade objekt. Det kan dock också vara ett specifikt objekt där man t.ex. inte än vet om det motsvarar en klass. Varje klass ska ha en tydlig och avgränsad uppgift som kan beskrivas med några få korta meningar (helst bara en enda). Klassnamnet skrivs längst upp på kortet. Klassbeskrivningen kan skrivas på baksidan av kortet. Det är viktig med bra namn, så att klassens uppgift är klar och tydlig utan att man behöva titta på klassbeskrivningen hela tiden.
- **Responsibilities:** ett ansvar är nånting objekten av en klass vet eller gör för att fylla sin uppgift i systemet. Ansvaren motsvarar tjänsterna som krävs från objekten. En bok i ett biblioteksystem borde t.ex. veta om den är utlånad eller försenad och när den ska lämnas åter.
- **Collaborators:** en samarbetspartner är en annan klass som behövs för att klara av ett ansvar. Samarbetspartnern delar på sätt och vis ansvaret genom att ställa till förfogande information eller ta över en del av ansvaret (genom sina egna ansvar). En bok kan t.ex. bara veta om det är försenad om det känner till aktuellt datum (vilket i exemplet är ansvaret av klassen `Datum`).
- Baksidan av kortet används för att definiera klassens syfte och kommentarer. Man kan också lista ytterliga ansvar där som inte ryms på framsidan.



CRC-kort är särskild lämpade för grupparbete. Det har visat sig att en gruppstorlek av 4-6 personer fungerar bäst. OBS! Det är viktigt att inte låta sig frestas att diskutera kodningsdetaljer i CRC sessionerna. Syftet är att ta fram, diskutera och testa olika analys respektive designmodeller.

3 OO-Analys

I normalfallet är ett problem alltid oklart och illa formulerat. Under analysen reder man ut vad som faktiskt skall göras, men också vad som inte ska göras. Vad är problemet egentligen? I denna fas försöker vi

- *hitta kandidatobjekt* med hjälp av brainstorming (avsnitt 3.1)
- *filtrera kandidatobjekten* (3.2)
- *skapa CRC-kort* för kandidatobjekten (3.3)
- *definiera scener* för testning av modellen (3.4)
- *förbereda gruppsessionen* (3.5)
- *"spela in" scener* med hjälp av CRC-korten (3.6)
- *dokumentera scener* med hjälp av sekvensdiagram (3.7)

3.1 Brainstorming

Idén är att försöka att på ett någorlunda okomplicerat sätt ta fram ett antal kandidatobjekt att ha som utgångspunkt för den fortsatta analysen.

Försök att skriva ner alla objekt du kan komma på som har relevans i problemområdet. Den enda regeln är att det du skriver ner skall vara ett *substantiv*[2], eftersom fokus skall ligga på objekt i det här skedet. Ett bra objekt skall ha både *egenskaper* och *beteenden*. Om två tänkbara objekt endast skiljer sig åt vad gäller värdet på deras egenskaper så måste de betraktas som samma slag av objekt, som vi sedan kallar klass.

Ett exempel: om Eva och Bosse skall representeras i vårt system men bara skiljer sig åt vad gäller t.ex. namn och adress så borde de betraktas som samma slag av objekt. Men om Eva är brandsoldat och Kalle är polis och vi bygger ett system för att hantera nödlarm så kommer Eva och Kalle att representera olika beteenden och borde därför båda vara kandidatobjekt. Så småningom är det rimligt att tänka sig brandsoldatsobjekt och ett polisobjekt istället för Eva och Kalle.

Som hjälpmedel i brainstorming fasen kan man använda sig av två knep för att "komma på" kandidater:

- En checklista med källor/områden för typiska objekt. Man kan t.ex. fundera på personer, platser, ting som kan röras vid, roller, organisationer, abstrakta ting, händelser, interaktioner, osv. som har att göra med problemet.
- Man kan också leta substantiv i problembeskrivningen för att hitta lämpliga kandidater.

OBS! I brainstorming fasen ska alla komma till tals. Förslagen ska inte diskuteras i denna fas. Rensa bland kandidaterna görs först i nästa fas. Lämpligen går man runt i gruppen och skriver upp alla förslag som görs på en synlig plats, t.ex. på en tavla.

3.2 Filtrera kandidatobjekten

Så är det dags att börja filtrera och sortera bland de funna kandidatobjekten. Målet är att

[2] Det kan också vara en s.k. nominalfras där flera ord tillsammans bildar en ordgrupp som kan användas som substantiv, t.ex. …

filtrera bort alla kandidater som inte beskriver problemområdesrelaterade objekt.

- Sortera först bort synonymer och saker som antingen ligger utanför systemet eller som beskriver detaljerna i implementeringen. Tittar sedan närmare på alla substantiv som egentligen är verber eller beskriver aktiviteter. Dessa är oftast inga bra kandidater. De borde hellre beskriver vad objekten gör, dvs. ansvarigheter (se avsnitten om CRC-kort). Likaså borde man se upp med substantiv som beskriver (mät)enheter och enkla egenskaper. De borde hellre beskriver vad objekten vet.
- Filtrera sedan bort de objekt som har med gränsytan/interfacet mot människan att göra från de problemområdesrelaterade objekten. Oftast har interface-objekten inget med förståelsen för problemet att göra. De allra flesta materiella objekt vi har omkring oss saknar interface. T.ex. håller man en penna eller en läskedrycksflaska på det sätt man gör på grund av deras form - inga speciella knappar eller handtag är nödvändiga. Det är den fysiska strukturen som är interfacet. Andra objekt har interface som är nödvändiga men som inte har med objektets grundläggande beteenden att göra. Kylskåp behöver ha ett handtag på dörren för att den ska kunna öppnas, men handtaget är inte inblandat i beteendet att hålla innehållet kallt. En radio har en volymkontroll, men radion måste ta emot en station och spela oavsett volymens aktuella inställning. Fokusera på kärnverksamheten, inte på hur någon framtida användare ska interagera med objektet.
    - Finns det objekt som egentligen är egenskaper hos andra objekt? T.ex. så är Namn sällan ett eget objekt, utan oftast en egenskap (ett attribut) hos t.ex. en klass Brandsoldat/Polis.
    - Är några av objekten egentligen mer detaljerade varianter av andra objekt? Det betyder inte att du ska kasta bort kandidatobjekt, men kan börja tänka på en struktur av potentiella objekt på ett tidigt stadium. En Kriminalinspektör är en Polisman t.ex.
    - Är några av kandidatobjekten egentligen instanser av något annat objekt? Tänk på generella objekt i detta skede även om det i slutänden bara kommer att finnas en instans av det givna objektet.

Slutligen gör du en sista filtrering av dina kvarvarande objekt. Vilka av dessa objekt är sådana att du verkligen kommer att jobba med dem? Några av kandidatobjekten hör egentligen hemma i ett annat problemområde. Om man t.ex. designar ett kassasystem skulle ganska enkelt kunna utvidgas till att uppdatera lagret, lägga in poster på konton och omedelbart informera huvudkontoret om varje enskild försäljning. Men även om ett bra system innehåller alla dessa delar, så fungerar det inte att försöka göra allt på en gång. Försök bestämma vad som är den minsta mängden objekt som behövs för att få den funktionalitet som behövs.

3.3 Skapa CRC-kort

För varje identifierad objekt/klass skriver vi nu ner ansvaren på ett CRC-kort. Med hjälp av dessa kan man senare utforska hur klasser interagerar med varandra och tillhandahåller service till varandra under det att man med "rollspel" genomför ett antal tänkbara situationer

Korrespondenskort fungerar bra som CRC-kort och det rekommenderas starkt att man verkligen använder fysiska kort eftersom dessa är enkla att manipulera och lätt kan delas av gruppen. Korten ifylls i enlighet med beskrivningen i avsnitt 2.

Hur ett visst ansvar skall kodas (i t.ex. Java) är inte intressant i det här läget, rätt attityd här är "det struntar jag i". Det är heller ingen katastrof att samma ansvar förekommer i flera klasser i det här skedet, det kan hjälpa dig att se tänkbara samarbetsmöjligheter mellan olika objekt.

Ansvar och samarbete hör alltså intimt ihop.

Ingen klass ska ha oproportionellt många ansvar. I en "bra" objektorienterad modell ska objekten/klasserna dela på ansvaren. Man ska kunna föreställa sig objekten som varelser som kan agera på eget behov. Det ska inte finnas ett centralt objekt som styr hela systemet.

3.4 Definiera scener

Nu ska du definiera scener - hypotetiska händelseförlopp i det framtida systemet. En scen kan liknas vid ett manus som objekten ska spela in. Syftet med scenerna är att "testa" om objekten kan hantera alla händelser och situationer på rätt sätt som kan uppträda när en scen spelas in. Typiska scener är att utvärdera "vad händer om" situationer.

Scenerna ska vara konkreta och väl avgränsade. Annars är det stor risk att man tappar bort sig när scenerna "spelas in" (se avsnitt 3.6). Det är lämpligt att dela upp långa eller komplexa scener i delar som spelas in var för sig. När man senare diskuterar mera invecklade scener kan man hoppa över de delar som man redan har klart för sig i detalj.

Om du t.ex. skulle designa ett system för kursregistrering så skulle följande scener vara intressanta:

- Studenten Kalle vill registreras på kursen OOPU, men uppfyller ej förkunskapskravet PVK.
- Studenten Eva vill registreras på kursen OOPU. Hon uppfyller förkunskapskraven, men det finns inga platser kvar på kursen.

3.5 Förbereda gruppsessionen

Inför "inspelningen" ska alla i gruppen vara överens om ansvaret för respektive klass. Gå gärna igenom alla kort en gång till. En person i gruppen ska vara sekreterare och antecknar händelseförloppen. Sedan fördelas CRC-korten bland resterande gruppmedlemmarna. Varje gruppmedlem får till uppgift att agera objekt av den tilldelade kortens sort. Man ska vara noga att bara agera i enlighet med ansvaren som finns nedtecknad på kortet och inte förutsätta någonting som objektet inte vet. Dessutom ska man bara agera när de "egna" objektens tur.

3.6 "Spela in" scener

Nu är det dags för gruppen att "spela in" scenerna en i taget. Nu simuleras på sätt och vis hur det framtida systemet fungerar, om det vore byggt enligt den nuvarande modellen.

För enkelhetens skull kan ni anta att det finns ett objekt som representerar systemets gränsyta mot användarna och ta hand om all användarinteraktion. Ni ska i detta skede inte bry er om detaljerna i användargränssnittet.

- Vilket objekt kommer att få det meddelande som startar upp denna scen? Lägg fram motsvarande kort på bordet, så att alla kan se det.
- Vad behöver objektet känna till och ha ansvar för? Var noga med att kontrollera att objektet verkligen har all information som krävs för att svara mot uppgiften. Om informationen inte finns nedtecknad så gör det nu[3]. Om "inspelningen" kör fast, t.ex.

[3] Det kan t.o.m. visa sig att hela kort saknas eller att flera kort är så pass lika att de borde slås ihop.

p.g.a. alltför mycket information som saknas eller om det visar sig att större ändringarna i modellen krävs, kan ni avbryta "inspelningen" och göra om modellen innan ni fortsätter.

- Vilka andra objekt behöver det samarbeta med? Allteftersom objekt berörs läggs motsvarande kort fram på bordet och de ansvariga för respektive kort ta över "inspelningen".
- Var noga med att inte hoppa över småsteg som du tycker är självklara. "Djävulen sitter i detaljerna" (som man brukar säga på Tyska). Det är viktig att noga gå igenom alla steg som krävs för att svara mot en uppgift.
- När ett objekt är klar med sin aktuella uppgift så tas dess kort bort från bordet. "Inspelningen" fortsätter med föregående kortet.

CRC-korten kan hjälpa till att kontrollera att alla delar av problemet täckts in och ge förståelse för hur objekten interagerar. Dessutom blir det enklare att upptäcka objektens ansvar. Några andra fördelar med CRC-korten:

- Korten utgör en dokumentation av den första design-idén. I ett stort projekt kan det vara värdefullt att kunna få en uppfattning om bakgrunden till vissa designbeslut.
- CRC-kort kan användas av andra än programvaruutvecklare eftersom det inte finns något speciellt språk eller någon notation man behöver behärska. Det kan också vara ett sätt att försäkra sig om att fler än utvecklarna är införstådda i problemlösningen.
- CRC-kort är ett bra hjälpmedel för att testa olika analys- och designmodeller där det finns olika uppsättningar med kort med varierande ansvar. På så sätt kan olika alternativ testas långt före kodningsfasen.

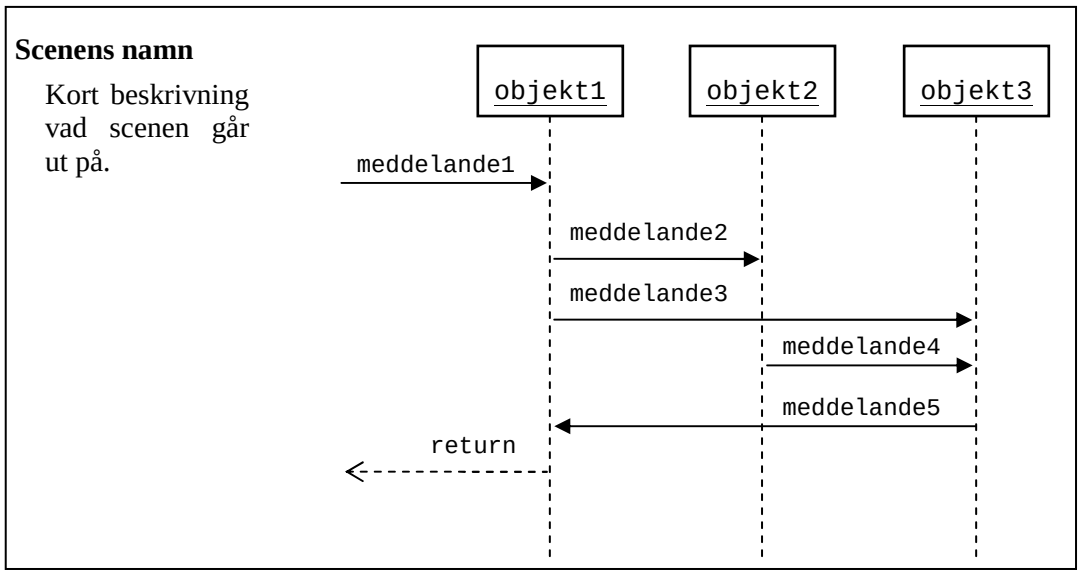
3.7 Dokumentera scener

Det är viktigt att dokumentera scenerna så att man senare kommer ihåg i vilken ordning saker och ting ska göras av vilka objekt. Dokumentationen har dock fler ändamål än så:

- När man har gjort ändringar på modellen kan man lätt kontrollera om en scen fortfarande fungerar som tidigare.
- När man ska förklara analysmodellen för andra kan man visa hur interaktionen mellan objekten fungerar.
- När man ska implementera modellen har man redan klar en del av dokumentationen.

Om man väljer sekvensdiagram som notation för dokumentationen får man dessutom ett bra underlag för implementeringen. I sekvensdiagram, som också kallas för object interaktions diagram (OID) beskrivs hur ett ansvar fördelas över flera objekt och i vilken ordning olika deluppgifter utförs.

I diagrammen ritar man in alla inblandade objekt från vänster till höger. Meddelanden ritas in i tidsordningen uppifrån (den första) och ned. Varje objekt får en "livlina" som visar meddelanden som kommer in och som går ut. Varje meddelande har ett namn, vanligtvis en uppgift som ett objekt ska utföra eller ett resultat som levereras.



4 OO-Design

Resultatet av analysen är en uppsättning CRC-kort, definierade med avseende på ansvar och samarbetspartners. Under designprocessen kommer dessa att bli klassdefinitioner eftersom det inte är de faktiska objektinstanserna i systemet som designas. Snarare är det det generella beteendet för klasser av dessa objekt som designas. Genom att göra om objekten till klassdefinitioner får vi ett mer generellt system som kan anpassas till förändringar.

Det som eftersträvas i designprocessen är en modell av problemets lösning, tillräckligt detaljerad för att skriva kod från. I den här presentationen preciserar vi resultatet av designen till:

- Ett klassdiagram som definierar egenskaper och beteenden för varje klass och som formellt identifierar relationer mellan klasserna (se 4.1).
- En detaljerad beskrivning av varje beteende

Det finns många olika former för att beskriva resultatet av designen. En vanlig standard som fastlagts kallas UML (*Unified Modeling Language*). UML är ett kraftfullt verktyg med notation inte bara för klassdiagram utan även för att beskriva en scen t.ex. (se UML reference card, t.ex. http://www.holub.com/class/oo_design/uml.pdf).

4.1 Relationer mellan klasser

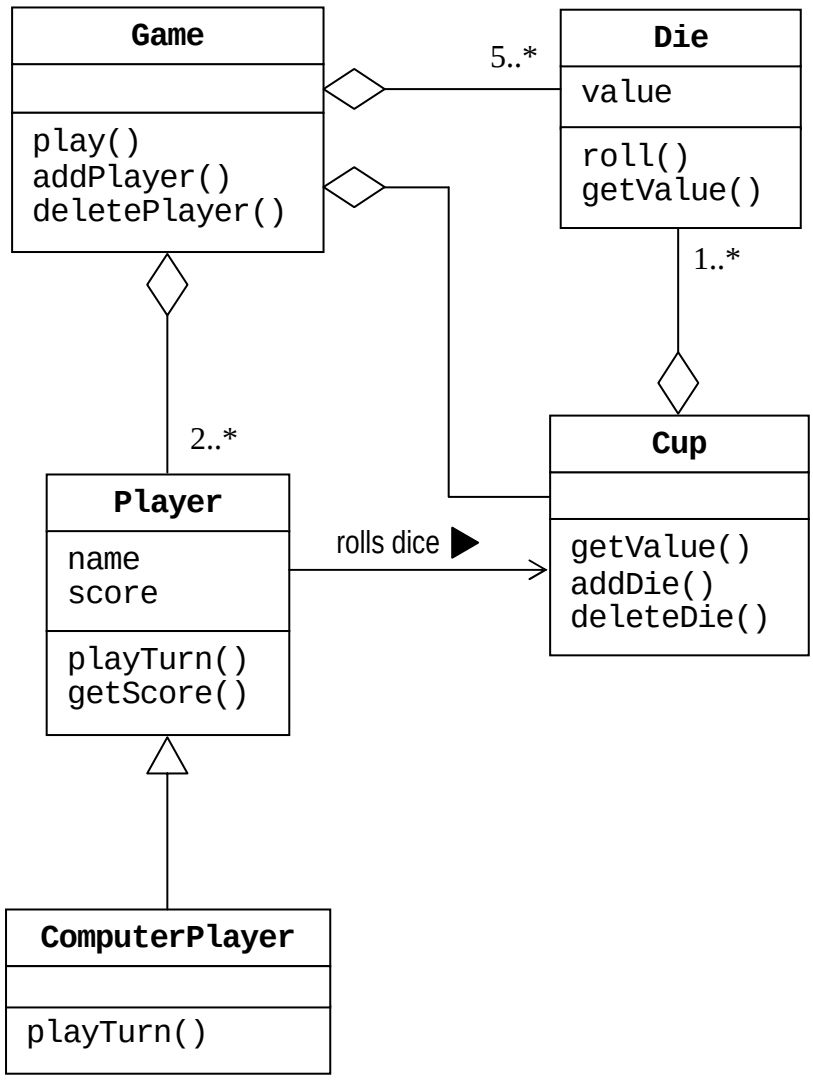
Relationer är alla de sätt på vilka två klasser kan interagera med varandra, de fyra grundläggande kategorierna är:

- **Generalisering:** kallas också Is-a eller Är-en. Denna relation indikerar att en klass är en mer generell/specialiserad variant av en annan klass.
- **Komposition/Aggregation:** kallas också Has-a. Denna relation indikerar att ett objekt är sammansatt av flera andra objekt. Det finns två varianter där delarna kan vara beroende eller oberoende av helheten. Om de är oberoende av helheten och "det hela" försvinner så förstörs inte delarna (aggregation).

- **Association:** denna relation indikerar en generell strukturell relation. Det är viktig att beteckna alla associationer med vettiga namn (oftast verb fraser).
- **Dependency:** kallas också Uses. Denna relation indikerar att det finns en relation, men detaljerna är än så länge oklart.

Generalisering är den starkaste och Dependency den svagaste relationen. UML har olika symboler för de olika relationerna vilka kan användas i klassdiagram.

I exemplet nedan indikerar den ofyllda diamanten en *has-a* relation, **`Game`** är en "helhet" som består av **`Player`**, **`Die`** och **`Cup`**. Om **`Game`** försvinner så kan dock övriga delar fortfarande existera (t.ex. i ett annat spel). Siffrorna indikerar vilken numerär relation klasserna har till varandra, **`Game`** har t.ex. minst fem tärningar, **`Game`** har också minst två spelare osv. Där siffra saknas är värdet ett. Den enkelriktade associationen (pilen) **`rolls dice`** anger att **`Player`** använder **`Cup`**, som i sin tur inte vet något om i vilket sammanhang den används. **`ComputerPlayer`** är en mer specialiserad slags **`Player`**. Den specialiserade varianten kan lägga till egna specifika egenskaper och beteenden eller definiera om ärvda beteenden. I exemplet omdefinieras metoden **`playTurn()`** (vi antar att en dataspelare spelar annorlunda än en vanlig spelare).



5 Fallstudie 1: en klocka

Som ett första exempel ska vi titta på hur man kan jobba för att ta fram designen för en klocka. Vi vill ha en bra, gammeldags okomplicerad klocka. Vi känner väl till klockor och är därför väl kvalificerade att göra analysen. Målet är att åstadkomma något som är lätt att underhålla och lätt att återanvända för att senare kunna använda det i andra sammanhang.

5.1 Hur man gör det snabbt … och fel

Först gör vi en "brainstorming" av kandidatobjekt

- En urtavla som man kan avläsa tiden på - vi lämnar frågan om digital eller analog visning öppen och även om det ska vara 12- eller 24-timmars visning.
- Intern tidtagning som ser till att klockan rör sig med lämpliga intervall.
- Förmodligen någon intern representation för timmar, minuter och sekunder.
- Något sätt att komma från den interna tidtagningen till sekunder och från sekunder till minuter och vidare till timmar.
- Någon typ av knapp för att ställa klockan.

Nu stannar vi här och filtrerar bland kandidatobjekten

- Urtavlan och knappen för att ställa klockan tar vi bort eftersom dom tillhör interfacet mot människan. Klockan måste visserligen ha ett sätt att presentera tiden och att låta användaren sätta tiden men det kan komma senare. Däremot måste det finna en möjlighet att utföra detta i klockan, dvs. klockan måste ha en sådan funktionalitet.

Övrigt tillsammans med delar för visning och sättning av tiden ser ut att vara en rimlig definition för en klass **`Clock`**. Vi kan identifiera ett flertal instansvariabler:

- för att hålla rätt på tiden: **`seconds`**, **`minutes`** och **`hours`**.
- för att bestämma hur tiden skall visas: **`displayFormat`**.

Vi kan också definiera ett antal metoder

- För att modifiera tidsvariablerna: **`getSeconds`**, **`setSeconds`**, **`getMinutes`**, **`setMinutes`** samt **`getHours`** och **`setHours`**.
- för att flytta fram en sekund: **`nextSecond`**.
- konvertera till visbart format: **`display`** och **`setFormat`**.
- kanske något som kan manipulera tiden i sin helhet: **`getTime`** och **`setTime`**.

5.2 Objektorienterad analys av klockan

Nu fick du inte möjlighet att fundera själv, men kan du se någonstans där vi kan ha resonerat fel?

- När tog vi beslutet att bara ha *en* klass, **`Clock`**? Att lägga allt i en och samma klass är ingen bra idé av flera skäl. Ansvar och auktoritet centraliseras vilket gör det svårare att arbeta i grupp på problemet och det drar inte fördel av objektorientering. Dessutom kommer det att bli svårt att återanvända klockan. Tänk på en vanlig klocka, t.ex.

klockradion som väcker dig på morgonen. Med all säkerhet finns det komponenter i den klockan (som hårdvarukrets och display) som lika gärna kan sitta i andra klockor (mikrovågsugnen t.ex.). På samma sätt är det bra om vår virtuella klocka byggs upp av delar som kan användas i många sammanhang.

- Vi började med informationen och listade alla attribut istället för att koncentrera oss på vad klassen ska göra och ännu tidigare vilket ansvar klassen har.
- Valet av begrepp och notation kan göra att vi omedvetet styrs mot ett givet språk (set/get, medlemsfunktioner osv.).

5.3 Brainstorming av klockan

Vi skulle kunna använda en del av den tidigare analysen, men börjar om från början ändå. Nu försöker vi istället tänka på relevanta *objekt*:

- En **`Display`** som har ansvaret för att visa tiden.
- En **`Time`** som har ansvaret för att hålla reda på timmar, minuter och sekunder samt deras inbördes relation.
- En **`Ticker`** eller **`SecondsTicker`** (om sekunder är den minsta enhet vi tar hänsyn till) som tillhandahåller en konstant tidpuls.
- En **`Clock`** som har ansvaret för att hålla reda på tiden och visa den på begäran.

Återigen så lämnar vi displayen tillsvidare eftersom det tillhör människa-dator interfacet. Resten verkar innehålla rimliga delar att börja med.

5.4 CRC-kort för klockan

Vad kan nu vara lämpliga scener/händelser att undersöka med CRC-korten.

- När **`SecondsTicker`** skickar ut en tidpuls måste en intern räknare uppdatera antalet sekunder, minuter och timmar vid behov.
- När en visning av tiden begärs måste det korrekta formatet för visningen bestämmas, tiden måste samlas ihop och (möjligen) konverteras till detta format.

Vi provar dessa två scener med CRC-kort. Först kommer en ny tidpuls.

- Kontrollen börjar hos **`SecondsTicker`**. Notera på CRC-kortet ansvaret att skicka ut en puls och att den skall samarbeta med **`Clock`**.

Class: *SecondsTicker*	
Responsibilities	Collaborators
Ge en tidspuls	*Clock*

- Nu kommer alltså **`Clock`** med i bilden. Den måste ta emot en puls från **`SecondsTicker`**. För det krävs egentligen ingen samarbetspartner - **`SecondsTicker`** initierar händelsen. Däremot skall **`Clock`** öka på representationen i **`Time`**. **`Clock`** samarbetar alltså med **`Time`**.

Class: *Clock*	
Responsibilities	Collaborators
Ta emot en impuls från SecondsTicker	
Informera Time om att en sekund har gått	*Time*

- När **`Time`** informerats om att en sekund har gått måste den uppdatera sin sekundrepresentation, som i sin tur kan medföra att minut och tim-representationen berörs. För att göra detta behöver **`Time`** inte samarbeta med någon annan.

Class: *Time*	
Responsibilities	Collaborators
Öka min representation av sekunder	
Om nödvändigt öka representationen för minuter och timmar	

Nu har vi definierat flera roller och aktioner mellan objekt. Vi tar nästa scen/händelse och ser vad som kan tillkomma. Nästa scen börjar med en förfrågan (från något externt objekt) om att visa tiden.

- När en tidsförfrågan kommer behöver **`Clock`** hämta tiden, så **`Time`** är en samarbetspartner. Observera att **`Clock`** 's andra ansvar och samarbetspartners består, i slutänden måste **`Clock`** designas för alla dessa ansvarsområden.

Class: *Clock*	
Responsibilities	Collaborators
Ta emot en impuls från SecondsTicker	
Informera Time om att en sekund har gått	*Time*
Visa tiden	
Hämta tiden	*Time*

- **`Time`** måste returnera tiden i ett sånt format att **`Clock`** kan manipulera den, eftersom det är **`Clock's`** ansvar att formatera den.

Class: *Time*	
Responsibilities	Collaborators
Öka min representation av sekunder	
Om nödvändigt öka representationen för minuter och timmar	
Returnera tiden i timmar, minuter och sekunder	

- **`Time`** kan lämna scenen efter att ha returnerat en rå representation av tiden och **`Clock`** formaterar den på lämpligt sätt för att returnera den till anropande objekt för visning.

Class: *Clock*	
Responsibilities	Collaborators
Ta emot en impuls från SecondsTicker	
Informera Time om att en sekund har gått	*Time*
Visa tiden	
Hämta tiden	*Time*
Formatera tiden till lämplig visningsformat	

5.5 Objektorienterad design för klockan

Nu har vi alltså CRC-kort som beskriver våra klasser. Vi har en rätt bra uppfattning om vad respektive objekt skall vara ansvarigt för och vad det *inte* har ansvar för, dvs. vad som skickas vidare till samarbetspartners. Nu kan vi alltså prata om vad objekten *vet* och *kan*.

- **`Clock:`** måste kunna sätta **`displayFormat`** (och alltså känna till det också) och returnera tiden i ett givet format. **`Clock`** behöver kunna reagera på **`nextSecond`** och vidarebefordra det till **`Time`**. **`Clock`** behöver alltså känna till **`Time`**.

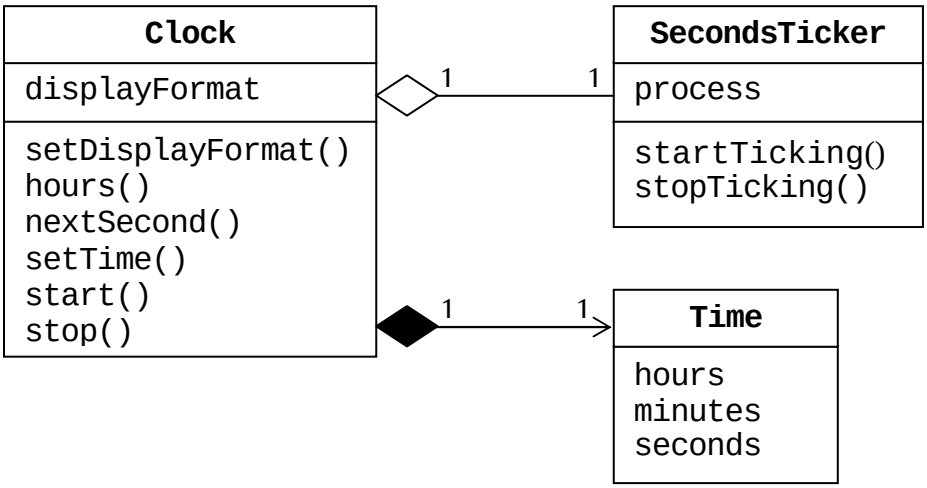
I enlighet med våra scener och vår CRC-korts analys behöver **`Clock`** inte samarbeta med **`SecondsTicker`** så **`Clock`** behöver alltså inte känna till detta objekt. Om man funderar lite kan man förstå att vi utelämnade en viktig scen (nu behöver vi alltså backa tillbaka till analysfasen och tänka om därifrån). Vad händer när klockan *startas*? När du startar klockan så är det egentligen klockans ansvar att sätta igång "tiden". Hur ska klockan kunna starta tiden om den inte känner till sin timer? Nu känns det mer motiverat att **`Clock`** känner till sin **`SecondsTicker`** med tanke på att **`Clock`** skall kunna startas och stannas. Starta och stanna klockan är i själva verket en förfrågan till timern att sluta ticka.

- **`SecondsTicker:`** måste känna till sin **`Clock`** för att kunna säga till när en sekund har gått. **`SecondsTicker`** måste kunna slås på och av. Förmodligen behöver den använda någon extern process för att generera tidssignaler, den behöver alltså känna till sin **`process`**.

- **`Time:`** måste kunna hålla reda på timmar, minuter och sekunder. Den måste kunna öka på antalet sekunder och även kunna låta denna addition spilla över på de andra enheterna.

Behöver inte känna till några andra objekt.

Vi karakteriserar förhållandet mellan **`Clock`** och **`SecondsTicker`** och mellan **`Clock`** och **`Time`** som associationer. **`SecondsTicker`** och **`Time`** *används* av klockan. **`Clock`** *har* dem. De är distinkta storheter men det är ändå tydligt att **`Clock`** är associerad med både en **`SecondsTicker`** och en representation av **`Time`**.



I vår modell vet **`Clock`** inte själv vilken tid det är, men den kan kolla up det genom att "fråga" sin **`Time`** objekt. Den fyllda diamanten indikerar att **`Time`** objektet försvinner när **`Clock`** objektet försvinner.

5.5.1 OOA för en väckarklocka

Nu borde det gå att skapa en väckarklocka **`AlarmClock`** eftersom den är en specialiserad variant av en klocka. Den stora skillnaden ligger i vad som händer vid den sekund när larmet ska "gå". Hur ska man upptäcka att tiden för alarmet är inne och vad ska göras?

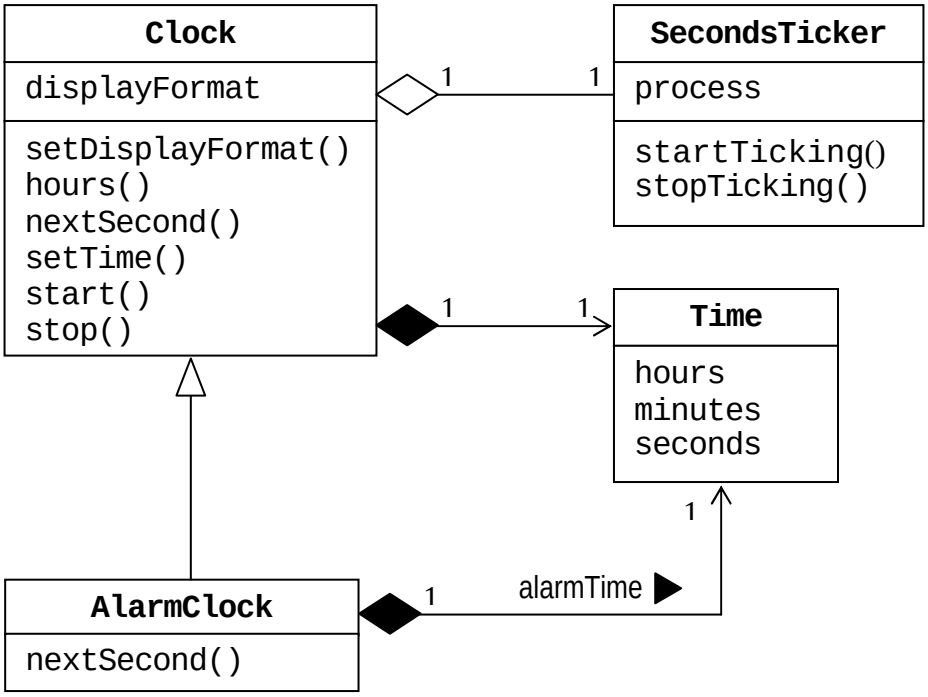
- **`AlarmClock`** ärver beteendet **`nextSecond`** från **`Clock`**.
- **`AlarmClock`** behöver ett **`Time`**-objekt för att representera larmtiden.
- När larmtiden är inne måste **`AlarmClock`** larma.

Class: *AlarmClock (is-a Clock)*	
Responsibilities	Collaborators
När en sekund har gått, gör den vanliga uppdateringen, jämför sedan min alarmtid med aktuell tid	*Clock, Time*
Om det är dags att larma, gör jag det som förväntas av mig	

5.5.2 OOD för väckarklockan

Nu försöker vi definiera **AlarmClock** mer noggrant. Den behöver ansvara för **nextSecond** själv (och inom sig anropa **Clock's nextSecond** beteende). Helt klart måste den känna till sin larmtid **alarmTime** så att **AlarmClock** äger (har) **Time**. Den behöver också veta mer precis

vilket dess beteende ska vara när "larmet går". Det modelleras med en relation **alarmTime** till **Time**. Så småningom borde vi kanske definiera en **Alarm**-klass som ansvarar för specifika beteenden i samband med ett alarm.



Det faktum att vi kan använda **Time**-klassen på mer än ett ställe (den löpande tiden i klockan och alarmtiden i väckarklockan) är en indikation på att designen håller bra så här långt. Ett bra test på den aktuella designen är möjligheten till framtida återanvändning. Om vi kan återanvända objekten i nya situationer så har vi lyckats åstadkomma definitioner av flexibla objekt.

Vidare användning skulle kunna vara att använda **AlarmClock** i en videobandspelare för att styra start och stopp av inspelning, eller i en almanacka för att hålla reda på bokade möten.

6 Fallstudie 2: ett bibliotek

Nu ska vi modellera ett lånesystem för ett institutsbibliotek. Det finns följande krav på systemet.

> I biblioteket finns böcker, tidskrifter, etc. Institutionens anställda ska kunna söka och låna saker från biblioteket. Alla saker i biblioteket har en registreringskod (ämneskod plus löpande nummer). Låntagare får låna upp till 10 saker.
>
> Sakerna kan bara lånas för en viss tid, beroende på vad det är för sak. Böcker t.ex. får lånas 6 veckor, men tidskrifter bara 3 dagar. För försenad återlämning debiteras en förseningsavgift, som f.n. ligger på 5:- per dag för böcker och 10:- per dag för tidskrifter.
>
> En låntagare får bara låna nya saker om denna inte har (1) lånat ut fler än 10 saker redan, (2) kvar saker som redan är försenade, (3) obetalda förseningsavgifter högre än 100:-.

I de följande delavsnitten ska vi gå igenom CRC metoden för biblioteksexemplet. I sista delavsnitt beskrivs ytterliga krav på systemet. Med dessa krav ska ni köra metoden ett (eller flera) varv till för att öva in den.

6.1 Hitta kandidatobjekt

En del möjliga kandidater finns (direkt eller indirekt) omnämnd i beskrivningen. Dessa är bibliotek, bok, tidskrift, institution, anställd, registreringskod, låntagare, lånetid, återlämning och förseningsavgift.

Genom brainstorming kommer vi fram till ytterligare kandidater som (t.ex.) bibliotekarie, användare och datum. Sedan finns ju i en bibliotek troligen fler typer saker att låna än bara böcker och tidskrifter, som t.ex. avhandlingar, konferensband, filmer, musik, programvaror eller dylikt.

6.2 Filtrera kandidatobjekten

Nu reder vi ut vilka av de föreslagna kandidaterna ska vara kvar. Det är viktig att komma överens om syftet och meningen med kandidaterna. Det kan därför vara en bra idé att personen som föreslagit en kandidat kort berättar varför den föreslagits. För enkelhetens skull gå vi nu igenom kandidaterna i tur och ordning.

- **Bibliotek:** samlingen av alla de objekt som finns i biblioteket (bokbestånd, tidskriftbestånd, osv.). Denna samling kan sökas igenom för att hitta saker som kan lånas ut. ⇒ OK.
- **Bok:** objekten som representerar böcker i biblioteket. ⇒ OK.
- **Tidskrift:** objekten som representerar tidskrifter i biblioteket. ⇒ OK.
- **Institution:** irrelevant, ligger utanför systemet. ⇒ ~~OK~~.
- **Anställd:** vad är skillnaden mellan anställd, låntagare och användare? Minst en av de är relevant för systemet. Systemet måste ju spara viss information om sina "kunder", t.ex. hur många böcker de har lånat och hur mycket obetalda förseningsavgifter de har. Information om personer som inte är (potentiella) låntagare behöver nog irrelevant för systemet. ⇒ ~~OK~~. Vi tar låntagare och förkastar anställd och användare.
- **Registreringskod:** finns det några specifika ansvar av registreringskoder? Hm, … de är nog bara en enkel egenskap hos böcker osv. ⇒ ~~OK~~; det är bara ett attribut hos bok, tidskrift, etc.

- **Låntagare:** se anställd. ⇒ OK.
- **Lånetid:** vad menas med det; den maximala tiden för utlåning (t.ex. 6 veckor för böcker), tiden (i t.ex. dagar) som återstår till återlämning, eller … Hur som helst kan vi inte komma på några specifika ansvar av lånetider. ⇒ ~~OK~~; det är nog bara ett attribut hos bok, tidskrift, etc.
- **Återlämning:** se lånetid. ⇒ ~~OK~~.
- **Förseningsavgift:** bara ett enkelt värde. ⇒ ~~OK~~; bara attribut hos låntagare.
- **Bibliotekarie:** det är klassen som har ansvaret för biblioteksfunktionerna som sökning, utlåning, återlämning, osv. Öh …, men varför behöver vi då ha kvar bibliotek? Bra fråga. Bibliotek objektet representerar bibliotekets innehåll (hyllorna så att säga). Bibliotek ansvarar dock inte för biblioteksfunktionerna. ⇒ OK. OBS! Vi borde kanske fundera på ett bättre namn för bibliotek objektet.
- **Användare:** se anställd. ⇒ ~~OK~~.
- **Datum:** i systemet behövs en hel rad datum som måste kunna räknas fram och jämföras, t.ex. för att kolla om lånetiden har gått ut. Det är en rad andra objekt som har användning av datumobjekt. ⇒ OK.
- **Avhandling, konferensband, film, musik, programvara:** dessa olika typer av saker som kan lånas ut har mycket gemensamt med böcker och tidskrifter (som t.ex. titel, författare, etc.), men de har också en del specifika egenskaper. För film, musik eller programvara behöver vi t.ex. få information om krav på plattform för att använda de. Det känns dock ohanterligt med så pass många olika klasser. Vi väljer därför att introducera en ny kandidat **publikation** som ta hand om alla de gemensamma ansvar av bok, tidskrift, avhandling, konferensband, film, musik och programvara. ⇒ ~~OK~~.
- **Publikation:** se ovan. Publikation är superklass för bok, tidskrift, avhandling, konferensband, film, musik och programvara. ⇒ OK.

Nu har vi följande objekt/klasser kvar: bibliotek, bok, tidskrift, låntagare, bibliotekarie, datum och publikation. Notera att det inte behövs några samarbetspartners för ansvaret att bara spara information. Samarbetspartners behövs bara om denna faktiskt ska göra något.

6.3 Skapa CRC-kort

Publikation: mängden av objekt som representerar saker som kan lånas från biblioteket.

Class: *Publikation*	
Responsibilities	Collaborators
vet om utlånad	
vet om försenad	
vet låntagaren	
beräkna tid för återlämning	*Datum*

Låntagare: mängden av objekt som representerar personer som lånar ut saker från biblioteket.

Class: *Låntagare*	
Responsibilities	Collaborators
får låna saker	
vet lånade publikationer	

Bibliotekarie: det objektet i systemet som ta hand om användarnas begäran att låna och återlämna publikationer och söka efter publikationer i biblioteket.

Class: *Bibliotekarie*	
Responsibilities	Collaborators
låna ut publikation	*Låntagare, Publikation*
lämna åter publikation	*Låntagare, Publikation*
sök efter publikation	*Bibliotek*

Datum: mängden av objekt som representerar datum i systemet.

Class: *Datum*	
Responsibilities	Collaborators
jämföra datum	
beräkna ny datum	

Bok: mängden av objekt som representerar böckerna i biblioteket.

Class: *Bok (is-a Publikation)*	
Responsibilities	Collaborators
beräkna tid för återlämning	*Datum*

Korten för bok, tidskrift, avhandling, konferensband, film, musik och programvara ser lika ut som för **Bok**. Vi presenterar därför bara **Bok** som ett exempel.

6.4 Definiera scener

Nu ska vi ta fram lämpliga scener. Det är viktigt att inte börja med en för generell scen av typ "vad händer om en person vill låna en publikation". Det skulle uppstå alltför många valmöjligheter att gå vidare i scenen och man skulle lätt tappa bort sig. Vad skulle t.ex. hända om personen ej är låntagare, eller om publikationen inte finns i biblioteket, eller om den är utlånad redan; eller om personen i princip får låna, men har redan lånat 10 böcker, eller ha obetalda förseningsavgifter, eller …?

Vi börjar med följande konkreta scen: "vad händer om Lennart Edblom vill låna boken med titeln *An Introduction to Computer Science Using Java*? LE har lånat en bok sedan tidigare som dock inte är försenad. Han har inga obetalda förseningsavgifter."

För att reda ut alla detaljer i ansvarsfördelningen måste denna scen gås igenom i flera varv.

I ett nästa steg kan vi prova några scener för återlämning av publikationer, t.ex. "vad händer om Lena Kallin Westin återlämna boken med titeln *Mathematical Logic for Computer Science* i tid?"

Hittills har vi antagit att allting går bra. Det är först nu vi tar oss an scener för att undersöka undantagen. Vad händer t.ex. om boken som LE vill låna redan är utlånad, eller om LE redan har lånat 10 böcker, eller om LKW har obetalda förseningsavgifter med 150:-?

6.5 Förbereda gruppsessionen och "spela in" scener

Vi börjar med scenen där LE vill låna Java boken som beskrivs ovan. Att spela in en scen innebär att vi i varje steg i scenen:

- identifiera närmaste ansvar (nästa deluppgift);
- identifiera objektet (kortet) som bär ansvaret (kan lösa deluppgiften); om ett sådant kort inte finns tilldela ansvaret det kortet som är bäst lämpade att bära ansvaret;
- låt personen som ansvarar för kortet redogöra i detalj hur objektet löser uppgiften (notera att detta kan innebära nya deluppgifter som andra objekt bär ansvaret för).

Vår scen börjar hos **`Bibliotekarie`**, eftersom det är objektet som bär ansvaret **`låna ut publikation`**.

Kortet `Bibliotekarie` *läggs fram på bordet.*

`Bibliotekarie:` för att låna ut boken *An Introduction to Computer Science Using Java* (JavaIntro) till LE måste jag kolla om LE får låna, dvs. uppfyller olika villkor. Jag själv har ingen kunskap om det. Ansvaret **`låna ut publikation`** har dock **`Låntagare`** som samarbetspartner och då kan jag fråga denne.

> *Kortet `Låntagare` läggs fram på bordet.*

`Låntagare (LE):` OK, jag har ett ansvar **`får låna saker`**. Detta gör jag genom att kolla upp om jag har lånat ut mindre än tio publikationer, vilket går bra då jag **`vet lånade publikationer`**.

> *Stop. Hur ska `Låntagare` veta att gränsen gå vid 10 lånade publikationer? Ansvaret för att känna till gränsen för antalet lånade publikationer ligger nog lämpligen i `Bibliotekarie`. Vi lägger alltså till ett ansvar `vet gränsen för antalet lånade publikationer` i `Bibliotekarie`.*

`Låntagare (LE) fortsätter:` då frågar jag **`Bibliotekarie`** om antalet publikationer som jag får låna.

`Bibliotekarie:` du får låna nya publikationer om du inte redan har mer än 10 lånade publikationer.

`Låntagare (LE) fortsätter:` OK, jag har ju bara en bok sedan tidigare. Sedan får inga av mina lånade publikationer vara försenade. Men det kan jag inte veta. Men jag skulle kunna fråga den lånade boken om **`får låna saker`** hade **`Publikation`** som samarbetspartner.

> *Detta verkar vara meningsfullt och vi lägger till `Publikation` som samarbetspartner för ansvaret `får låna saker`.*

`Låntagare (LE) fortsätter:` OK, då frågar jag min lånade bok om den är försenad.

> *Kortet `Bok` läggs fram på bordet.*

`Bok (den lånade):` OK, jag ärver ansvaret **`vet om försenad`** från **`Publikation`**. Jag är inte försenad.

> *Stop. Hur ska det gå till? Hur kan boken veta det? `Bok` vet ju ingenting om tider och datum och ännu mindre om hur man jämför olika datum. Vi måste i alla fall samarbeta med `Datum` för att kunna kolla om en försening har skett. Vi lägger alltså till `Datum` som samarbetspartner för ansvaret `vet om försenad`. Vi fortsätter dock med vår scen, men notera att detta måste redas ut senare. Kortet `Bok` tas nu bort från bordet och ansvaret går tillbaka till `Låntagare` objektet/kortet.*

`Låntagare (LE):` nu måste jag kolla upp om jag har obetalda avgifter. Detta vet jag dock ingenting om.

> *Det känns rätt att lägga detta ansvar till `Låntagare` också, eftersom det är information som berör själva låntagaren mest. Vi lägger alltså till ett ansvar `vet obetalda utgifter` på `Låntagare`.*

`Låntagare (LE):` OK, mina obetalda utgifter är 0, då får jag låna.

> *Kortet `Låntagare` kan nu tas bort från bordet och ansvaret går tillbaka till `Bibliotekarie` objektet/kortet.*

`Bibliotekarie:` nu vet jag att LE får låna, men jag måste också kolla upp om JavaIntro boken kan lånas, dvs. att den inte är utlånad. Det måste nog boken själv veta.

Kortet `Bok` läggs fram på bordet.

`Bok (JavaIntro):` OK, jag ärver ansvaret **`vet om utlånad`** från **`Publikation`**. Jag är inte utlånad.

Kortet `Bok` tas bort från bordet och ansvaret går tillbaka till `Bibliotekarie`.

`Bibliotekarie:` LE får låna JavaIntro. För att genomföra utlåningen måste boken beräkna tiden för sin återlämning och spara ned att den har lånats ut av just LE. Dessutom måste LE lägga till JavaIntro i sin "lista" av lånade publikationer. Allt detta går via ansvaret **`låna ut publikation`** som finns i **`Bibliotekarie`** som har just **`Låntagare`** och **`Publikation`** som samarbetsparters.

Vi börjar med bokens uppgifter och kortet `Bok` läggs fram på bordet.

`Bok (JavaIntro):` OK, jag ärver ansvaret **`beräkna tid för återlämning`** från **`Publikation`**, men hur det ska göras i detalj vet jag inte. Jag vet inte ens hur länge jag får lånas. Dessutom vet jag inte hur jag ska veta vem som håller på att låna mig.

Det var nog lite för mycket på en gång. Vi backar och ta det om. `Publikation` måste i alla veta hur länge den får lånas ut. Vi lägger alltså till ansvaret `vet maximum lånetid` i `Publikation`. När en publikation lånas ut måste den också kunna ändra på en del saker som den vet och det är inte bra om `Bibliotekarie` måste känna till alla dessa saker. Det är därför bättre om `Bok` ta över huvudansvaret för detaljerna i utlåningen, t.ex. genom ett ansvar `låna ut till (låntagare)` som sköter hela utlåningen under förutsättningen att låntagaren anges. Vi lägger alltså till ansvaret `låna ut till` med `Låntagare` som samarbetspartner till `Bok` (eller hellre sagt `Publikation`, eftersom samma resonemang gäller andra publikationer också) och börja om med `Bibliotekarie`.

`Bibliotekarie:` nu behöver jag bara säga till **`Bok`** att den ska **`låna ut till (LE)`**.

Kortet `Bok` läggs fram på bordet.

`Bok (JavaIntro):` nu kan jag veta att LE är ny låntagare och spara ner detta. Sedan vet jag nu att jag är utlånad. När jag har beräknat tid för återlämning borde jag också kunna spara ner tiden.

Vi uppdaterar ansvaren i `Publikation` enligt ovan.

`Bok (JavaIntro):` nu måste jag säger till LE att denne lägger till mig i sin "lista" av lånade publikationer.

Kortet `Låntagare` läggs fram på bordet.

`Låntagare (LE):` jag vet mina lånade publikationer men jag kan inte ändra i de.

Vi lägger till ansvaren `låna (publikation)` och `återlämna (publikation)` i `Låntagare`.

`Låntagare (LE):` OK. Nu kan jag uppdatera "listan".

Kortet `Låntagare` tas bort från bordet och ansvaret går tillbaka till `Bok`.

`Bok (JavaIntro):` jag är klar förutom detaljerna i **`beräkna tid för återlämning`**.

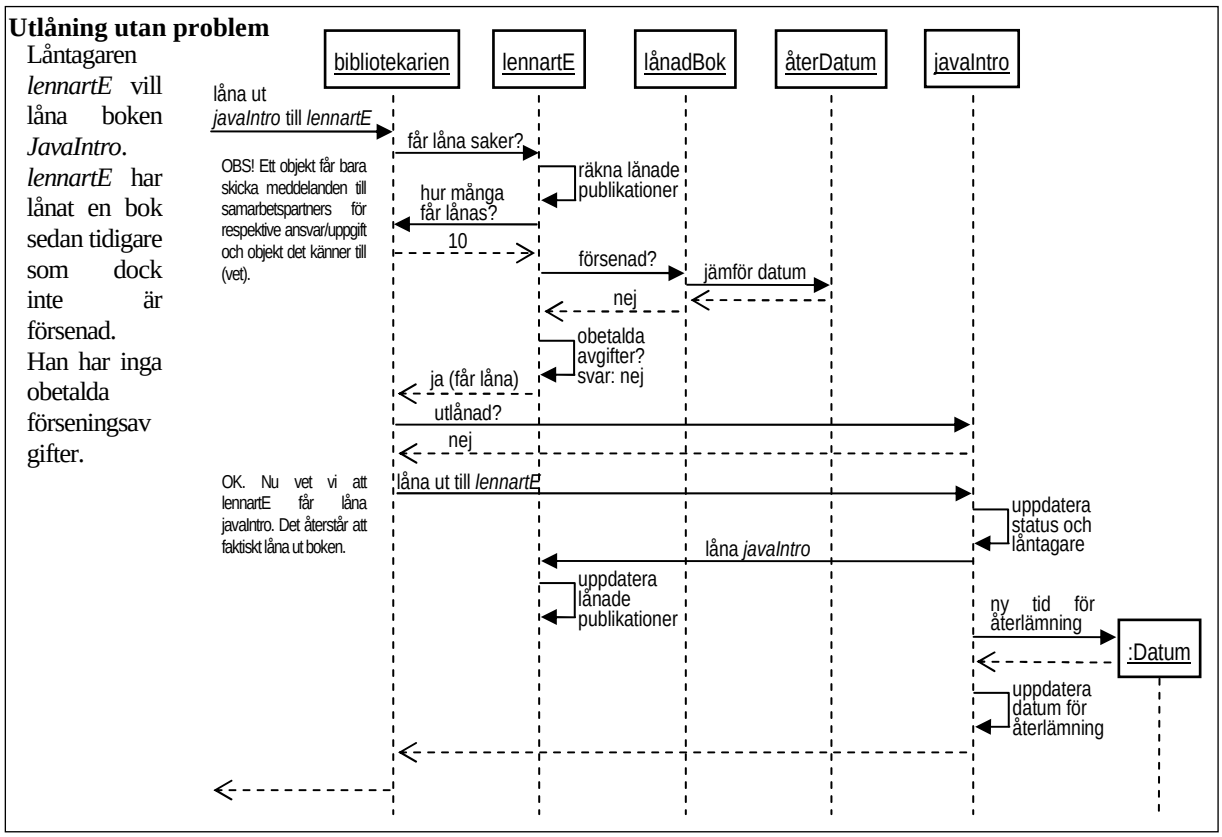
Detaljerna (som involverar `Datum`) tar vi senare. Kortet `Bok` tas bort från bordet och ansvaret går tillbaka till `Bibliotekarie`.

`Bibliotekarie:` jag är klar. LE har lånat den valda boken.

Hela scenen kommer att demonstreras "live" i föreläsningen.

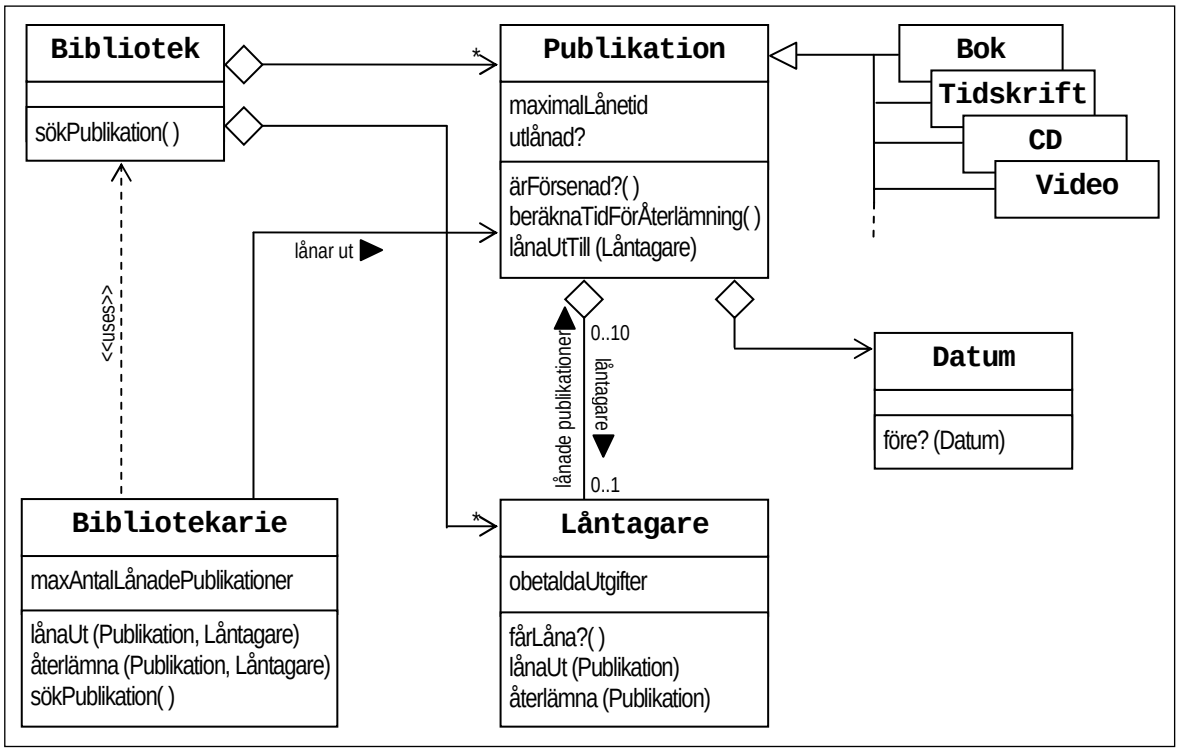
6.6 Dokumentera scener

Sekvensdiagrammet nedan dokumenterar föregående inspelningen i detalj.



6.7 Klassdiagram

Våra uppdaterade CRC kort kan "översättas" till följande klassdiagram.



6.8 Nya krav

I ett senare skede kan man tänka sig att kraven på biblioteksystemet höjs. Ta nu fram korten som har skapats så här långt och fortsätt med CRC metoden. I följande avsnitt beskrivs kraven som tillkommer.

> Om en sak är utlånad kan man skriva sig på en väntelista för saken. Saker med en tom väntelista får lånas om (dvs. man får förlänga lånetiden). Man ska också kunna registrera nya saker i biblioteket. Systemet ska då generera en registreringskod för saken. Det ska också vara möjligt att definiera nya ämneskoder.

Lycka till.

Källförteckning

Booch G., Rumbaugh J. och Jacobson I.: *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

Bellin D. och Suchman Simeone S.: *The CRC Card Book*, Addison-Wesley, 1999.

Guzdial M.: *Squeak—Object-Oriented Design with Multimedia Applications*, Prentice Hall, 2001.

Wilkinson N.M.: *Using CRC Cards*, Prentice Hall, 1995.