IN2001: Kravhåndtering, modellering, design

30 januar 2018

Yngve Lindsjørn

ynglin@ifi.uio.no

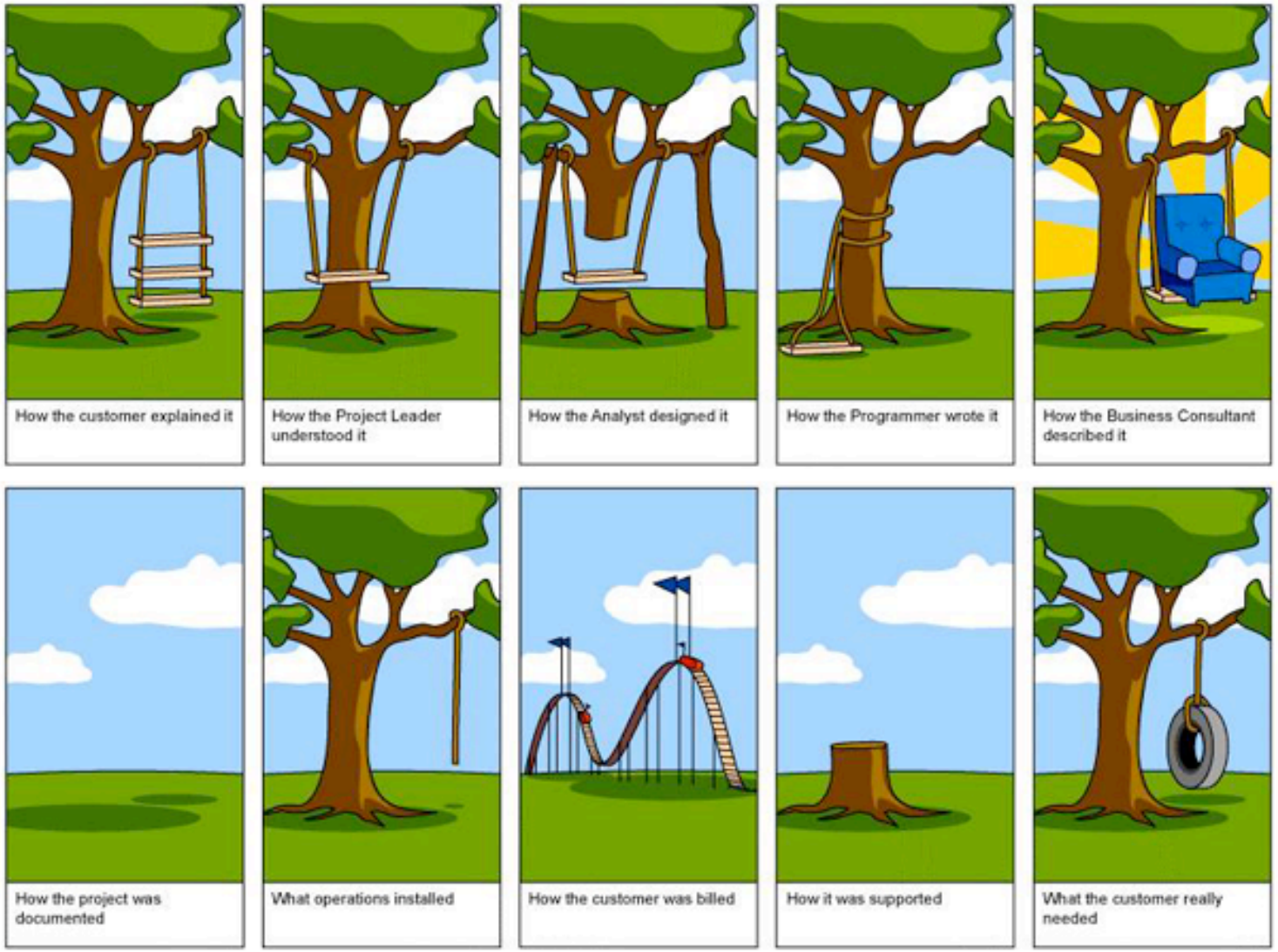
Gode beskrivelser av krav

er viktig for

- kontrakt oppdragsgiver – leverandør
- planlegging og oppfølging
- arkitektur, design og test
- å støtte videreutvikling og vedlikehold

Utfordringer i kravhåndtering

- Kommunikasjon
- Felles forståelse



Funksjonelle krav

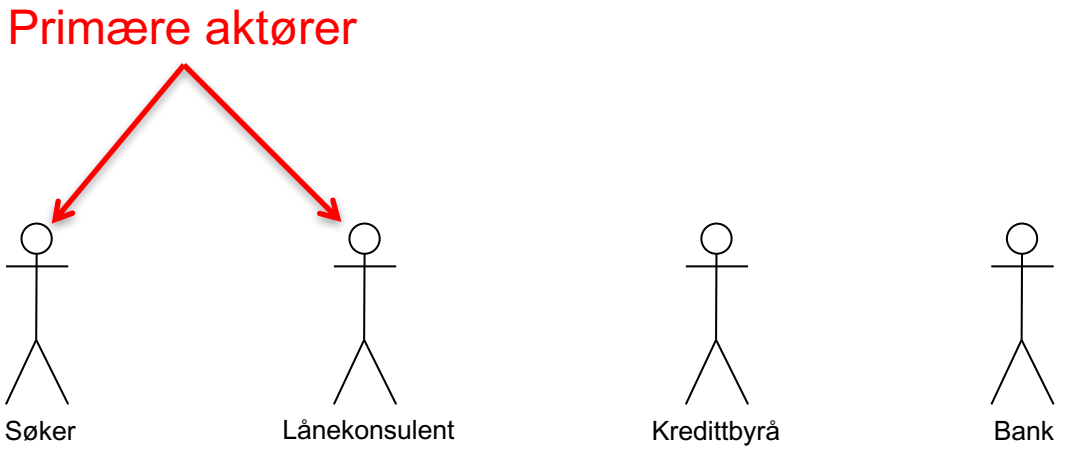
- Hva (ikke hvordan)
- Brukerkrav
  - Hva brukerne ønsker å kunne gjøre
- Systemkrav
  - Mer detaljert definisjon av hva som skal implementeres sett fra brukernes perspektiv

Beskrive krav

- Tekst
- Strukturert tekst
  - User story (brukerhistorie)
  - Use case (brukstilfelle)
- Modeller
  - UML (Unified Modeling Language)
  - BPMN (Business Process Model and Notation)

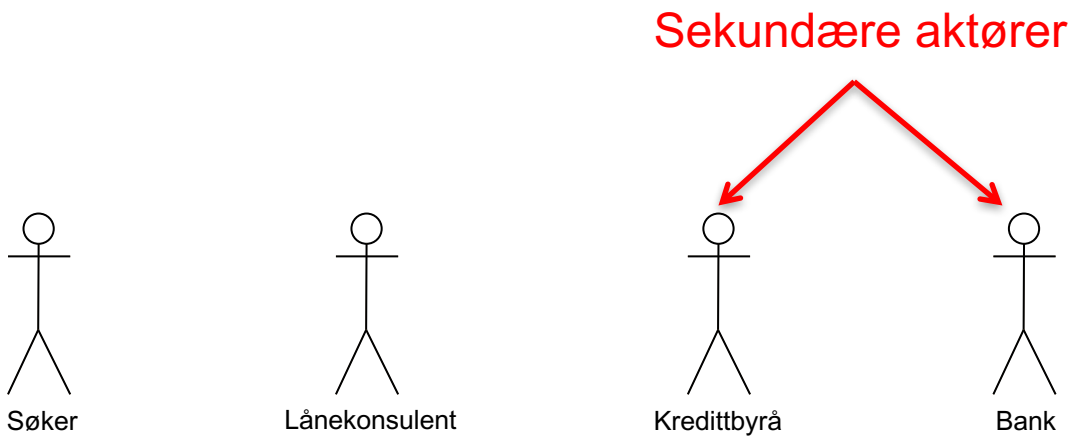
Use case modellering
Identifiser primære aktører

- Primære aktører har egne mål, dvs. de initierer use case (en eller flere) som oppfyller deres mål.



Use case modellering
Identifiser sekundære aktører

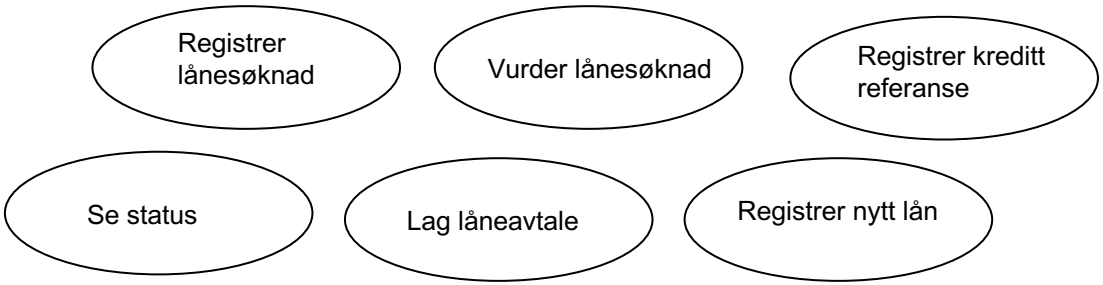
- Sekundære aktører har ikke egne mål, men er nødvendige for å realisere målene til de primære aktørene



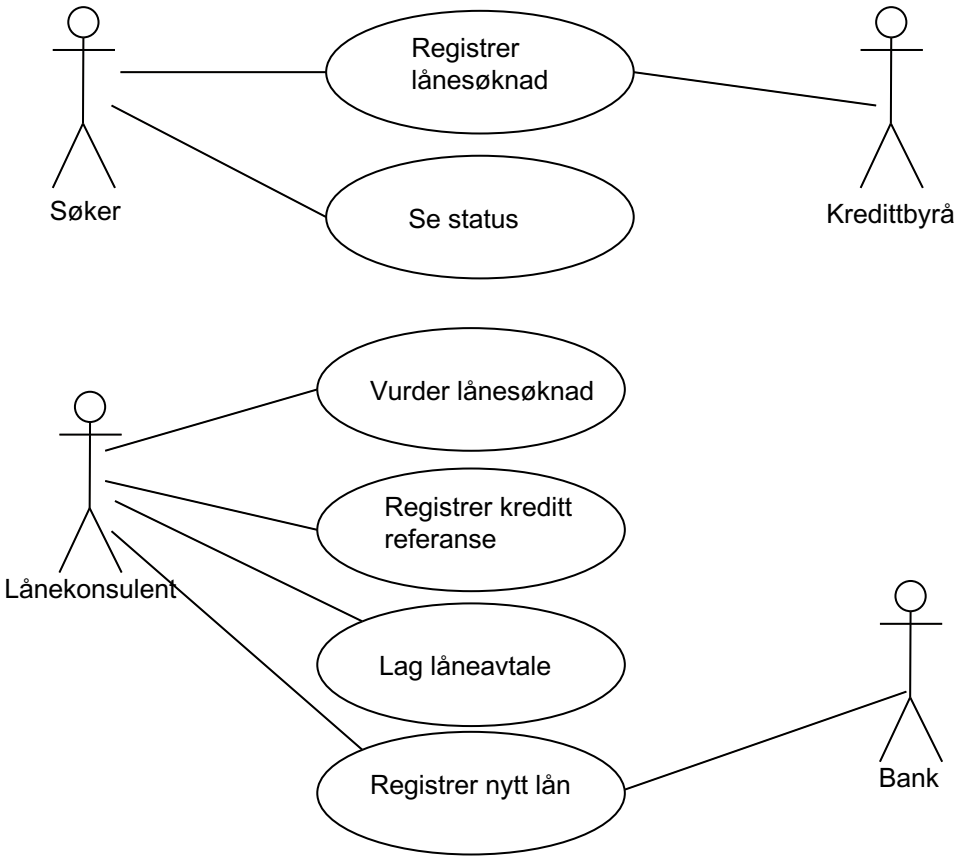
Use case modellering: Identifiser aktørenes mål → Use case

- Et use case beskriver hvordan systemet oppnår et mål av verdi for en aktør
  - En historie
  - Et komplett use case består av flere ulike hendelsesforløp (flyt)
- Et use case beskriver en komplett funksjonell enhet
  - One person – one place – one time
- Et use case er testbart

Eksempel: Use cases for lånesystemet



Use case modellering: Tegn use case diagram

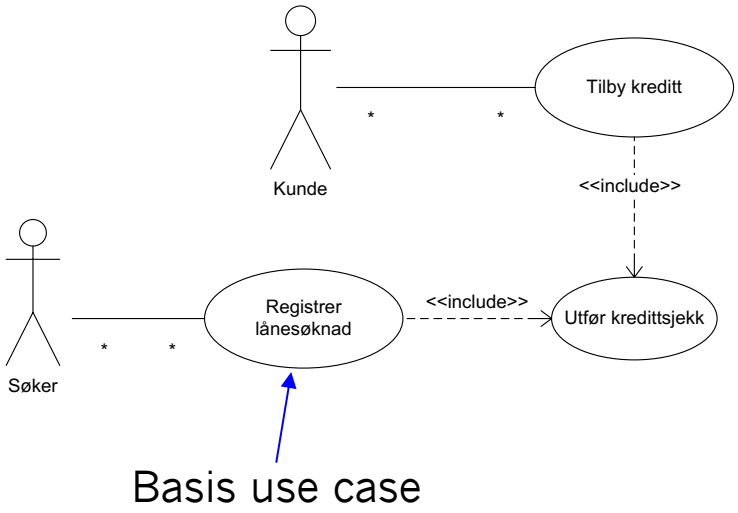


Relasjoner mellom og use case – Extend og Include relasjonen

- Include-relasjonen: Et use case kan være en del av ett flere andre use case.
- Extend-relasjonen: Et use case som beskriver tilleggsoppførsel som utføres under gitte omstendigheter

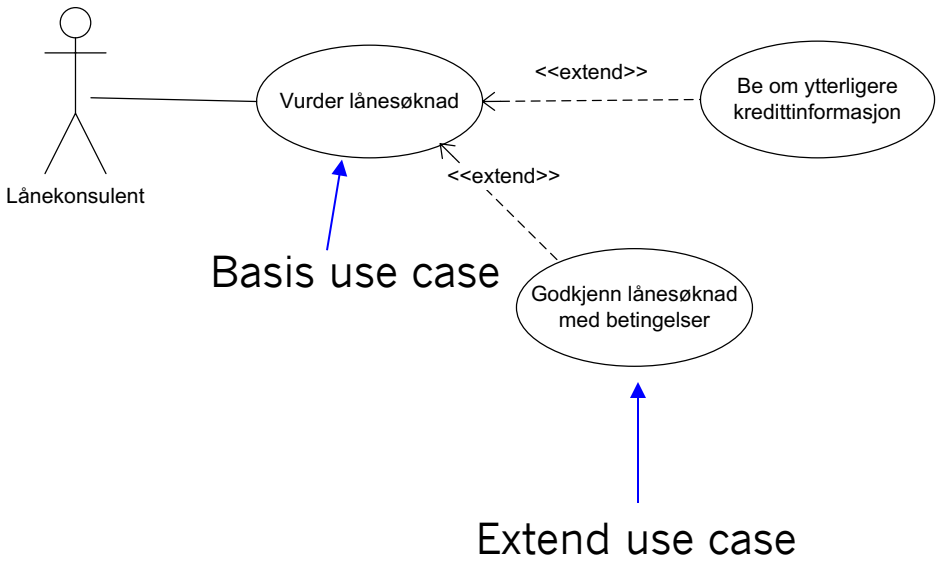
Include-relasjonen

- To eller flere use cases kan ha en felles del (noen like steg). Denne delen kan da legges ut i et eget use case som disse use casene kan inkludere.
  - Include kan også brukes for å forenkle store use case med mange steg
  - Include kan også brukes for å håndtere steg som kan forekomme når som helst i utførelsen av use caset
- Basis use caset vet hvilke use case det inkluderer



Extend-relasjonen

- Alternativ oppførsel som utføres i noen tilfeller kan skrives som eget use case som utvider (extends) et annet
- Extend use case beskriver hvordan oppnå et tilleggsresultat
- Basis use caset er fullstendig definert uten extensions, disse utvider funksjonaliteten
- Basis use caset kjenner sine extend use cases
- Bruk av alternativ flyt vs. bruk av extend use case:
  - Alternativ flyt beskriver hva som skjer ved avvik i normal flyt, mens
  - Extend use case beskriver hvordan oppnå tilleggsresultat.

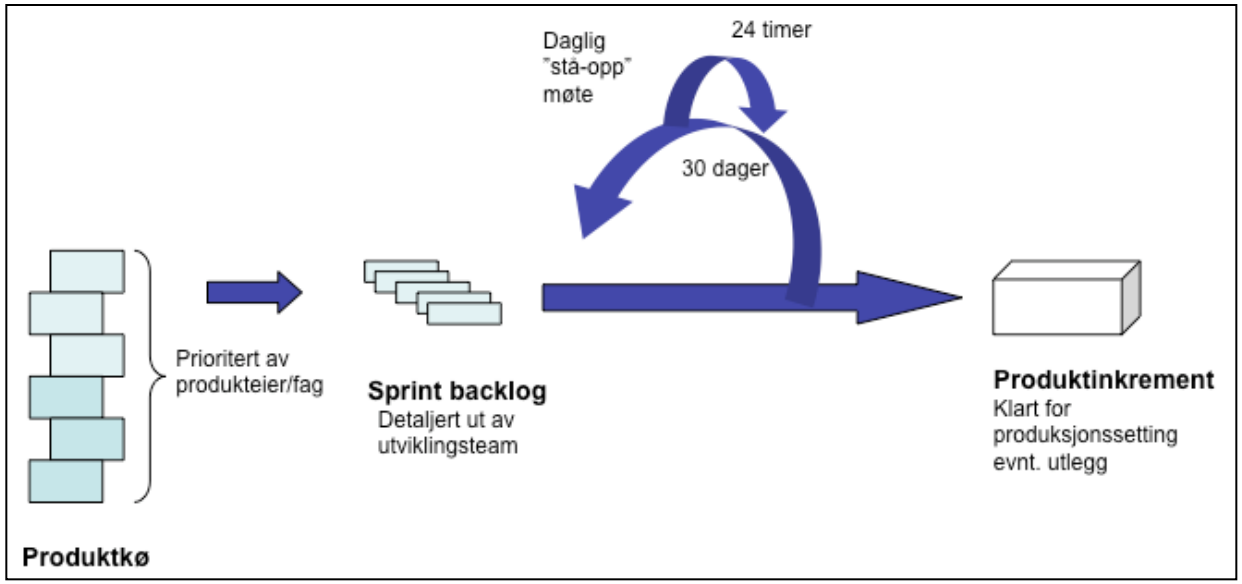


Use case vs. smidig utvikling

- I smidig utvikling jobber produkteier sammen med utviklere i samme team
- Det er mindre behov for detaljerte beskrivelser av krav og ofte brukes user stories (en ”lett” versjon av use case)

  Eks: **Som** lånekonsulent
  ønsker jeg å kunne vurdere lånesøknader
  slik at jeg kan gi en riktig og rask vurdering

- Krav utvikles underveis og beskrives ”on demand”
  - ❖ Først tilstrekkelig for prioritering i produktkøen
  - ❖ Så tilstrekkelig for prioritering i sprint backloggen



Use case vs. user stories

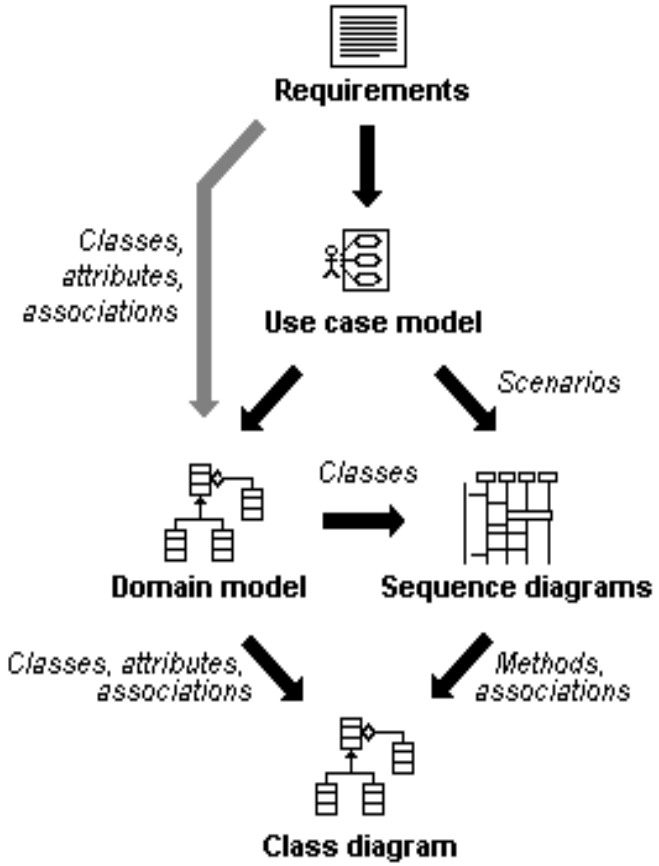
- Likheter. Begge viser
  - Hvem som skal bruke systemet
  - Hva de skal gjøre med det
  - Hvorfor de skal gjøre det
- Forskjeller
  - Omfang, kompletthet, livslengde, hensikt
  - User stories er godt egnet for å finne krav og bruke disse i smidig utvikling i samarbeid med produkteier/kunde
  - Use case er mer detaljert, har flere bruksområder videre i prosjektet og er mer egnet som dokumentasjon
- Men, det er en flytende overgang mellom dem.

Use case i prosjektplanlegging

- Planlegg hvilke use case som skal implementeres i hvilke iterasjoner av prosjektet:
  - Implementer use casene i henhold til hvor viktige de er og/eller hvor vanskelige de antas å være å implementere.
  - Hovedflyt implementeres først, deretter alternativene.
  - Estimer hvor mange use case (eller hendelsesflyt og alternative flyt) som kan implementeres i en iterasjon.

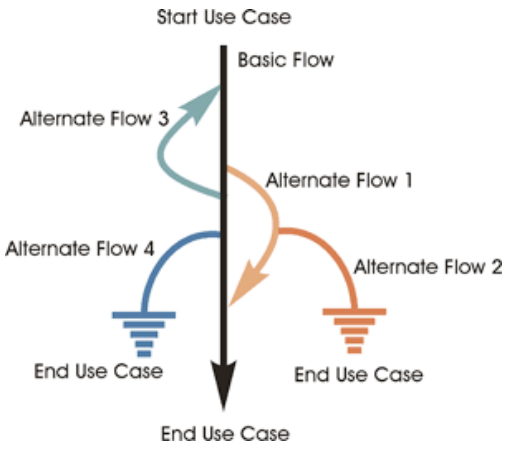
Use case i design

- Hendelsesflyt i use casene detaljeres ut i sekvensdiagram
- Domenemodellen utvides til klassediagram med systemklasser



Use case i funksjonelle tester

- Use case kan være utgangspunkt for testprosedyrer (manuelle eller automatiserte)
- Dette gir
  - Effektiv utforming av tester
  - Fokus på test av egenskaper som er viktig for bruker



Et testcase for hver vei Gjennom use caset

Testcase:

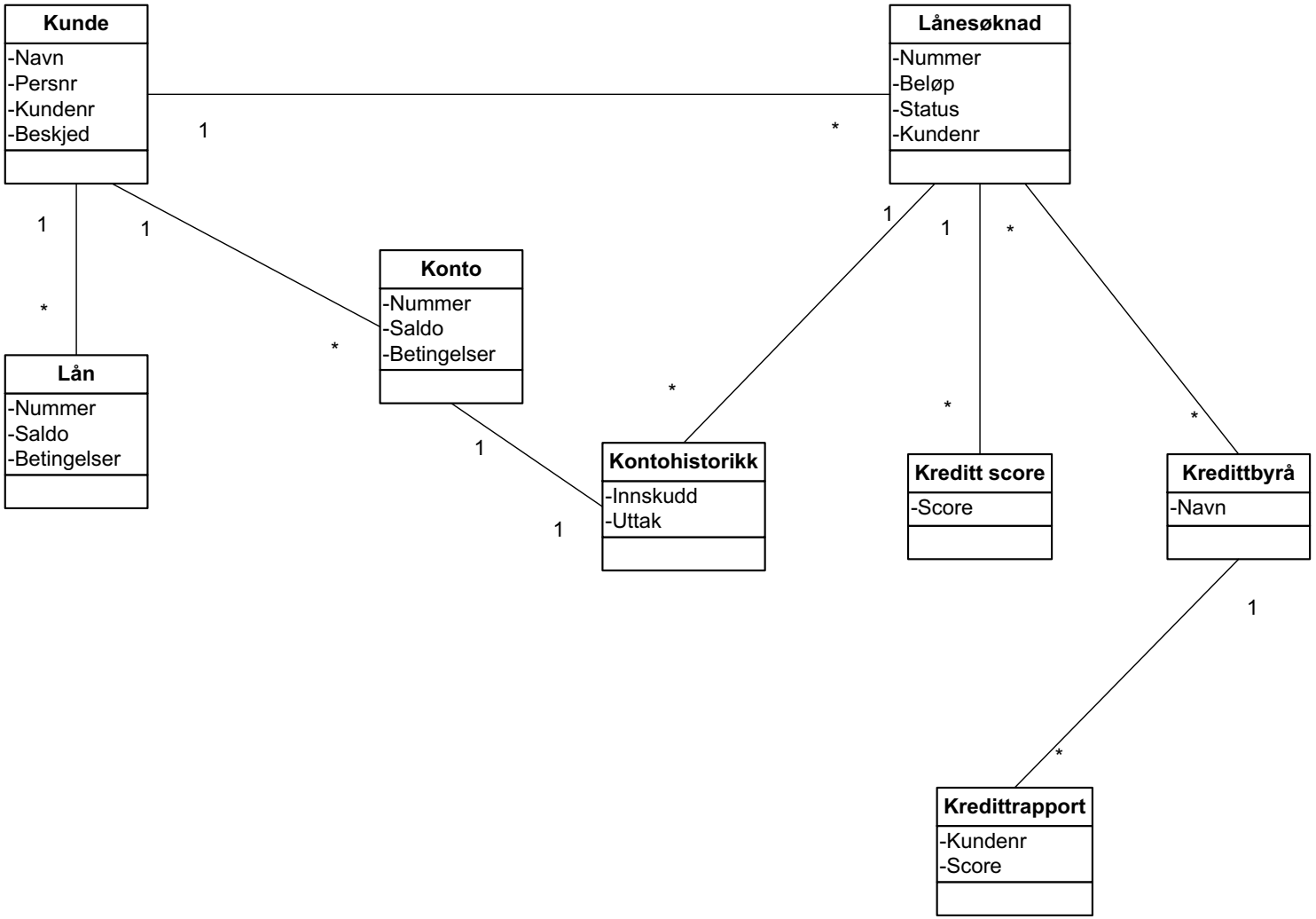
1. Prebetingelse
2. Input data
3. Steg
4. …
5. ..
6. Forventet resultat

Domenemodell

- UML klassediagrammer uten metoder

- Domenemodellen viser objekter i problemdomenet.
- Hensikten med domenemodellen er å forstå objektene og få en oversikt over terminologi.
- Domenemodellen er nyttig i forbindelse med use case modellering fordi:
  - Domenemodellen viser informasjonen om objekter i use casene.
  - Den er et viktig verktøy for å sjekke at use casene er beskrevet med riktig detaljeringsnivå.

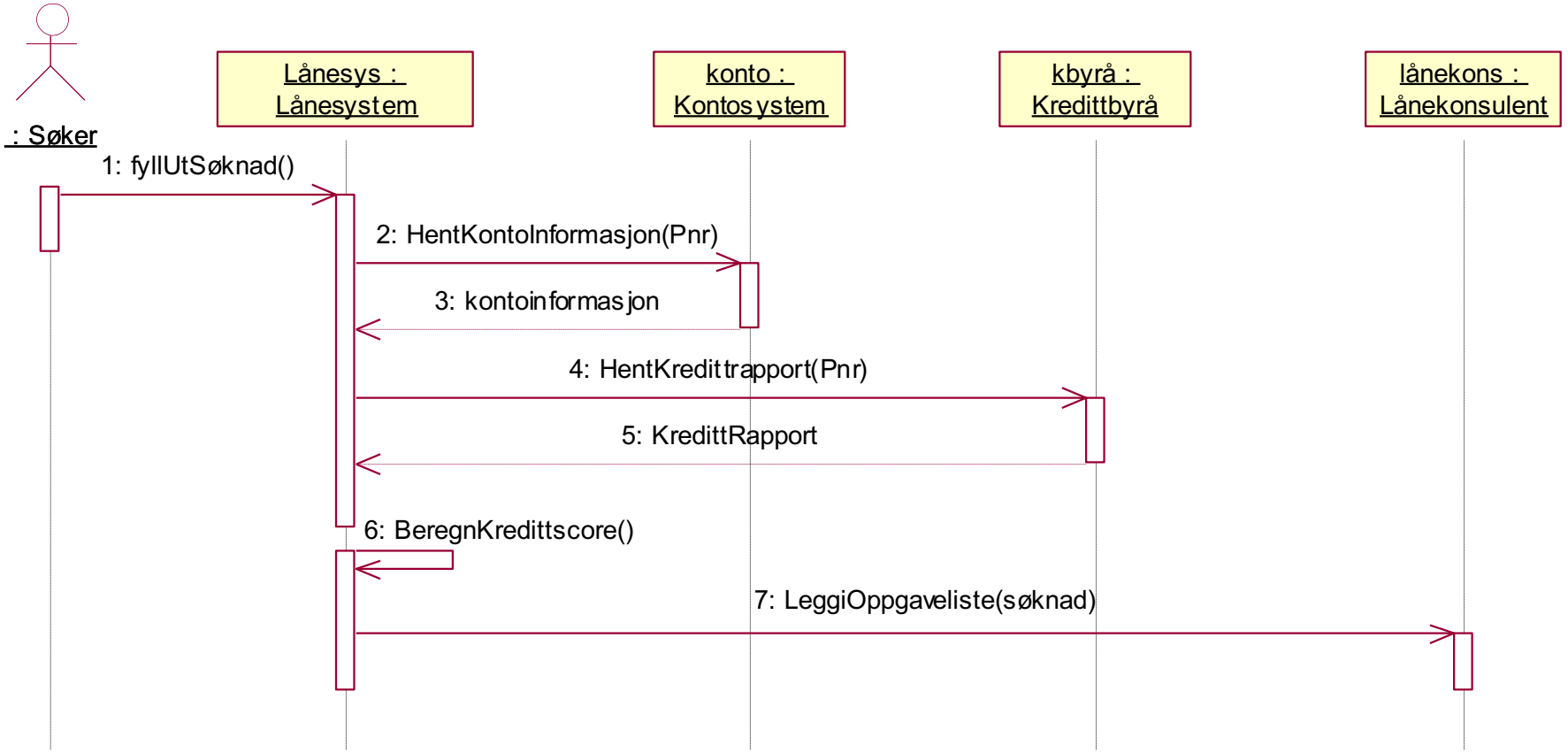
Domenemodell - eksempel



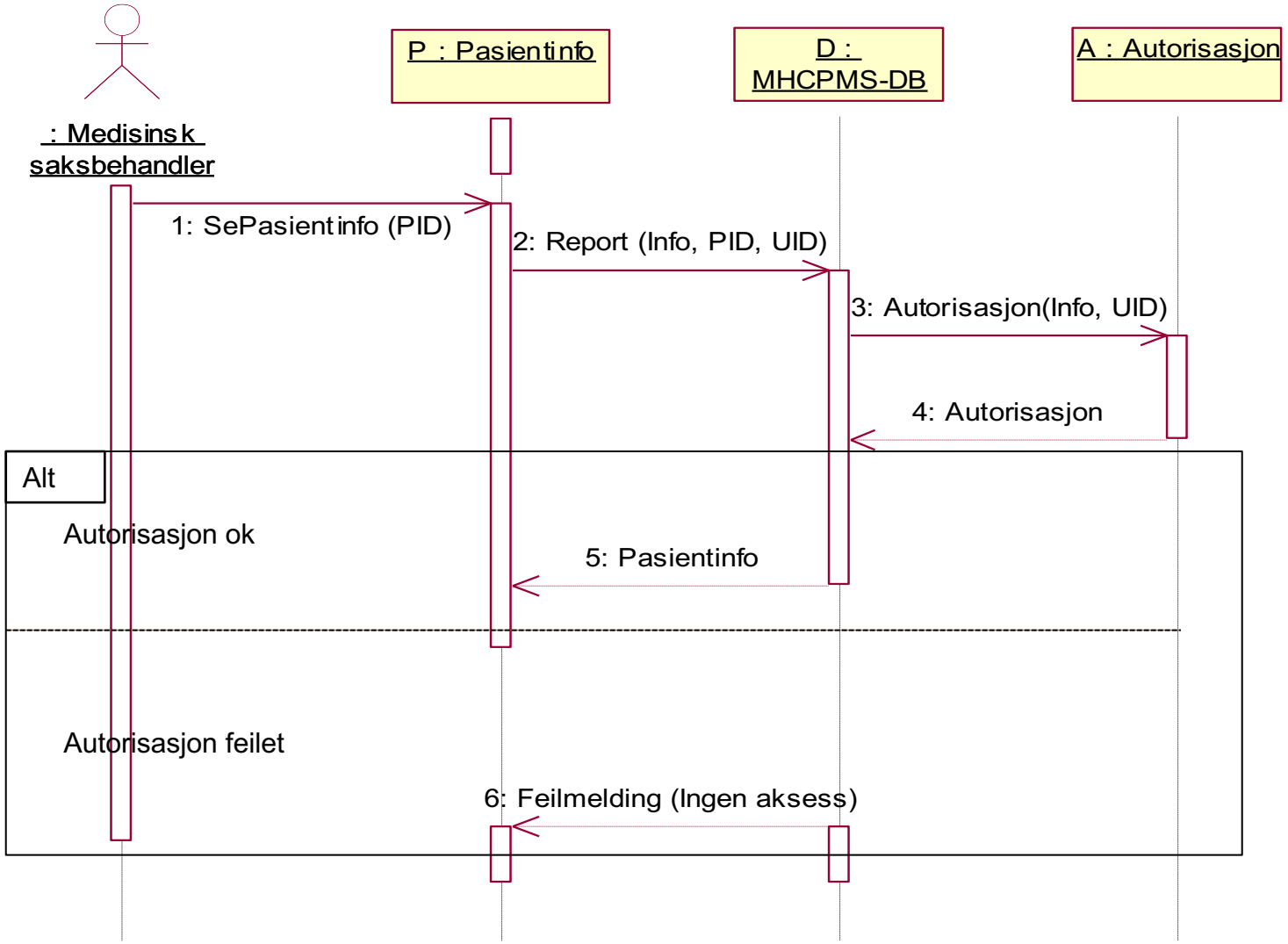
Sekvensdiagrammer

- Et flyt i et use case kan modelleres med sekvensdiagrammer.
- For hvert use case lages typisk sekvensdiagram for hovedflyt og for hyppig forekommende alternative flyt.
- Stegene (sekvensene – se tekstlig beskrivelse) i et use case vises som meldinger som sendes mellom objektene ved kall på objektenes metoder.

Sekvensdiagram “Registrer lånesøknad – hovedflyt”



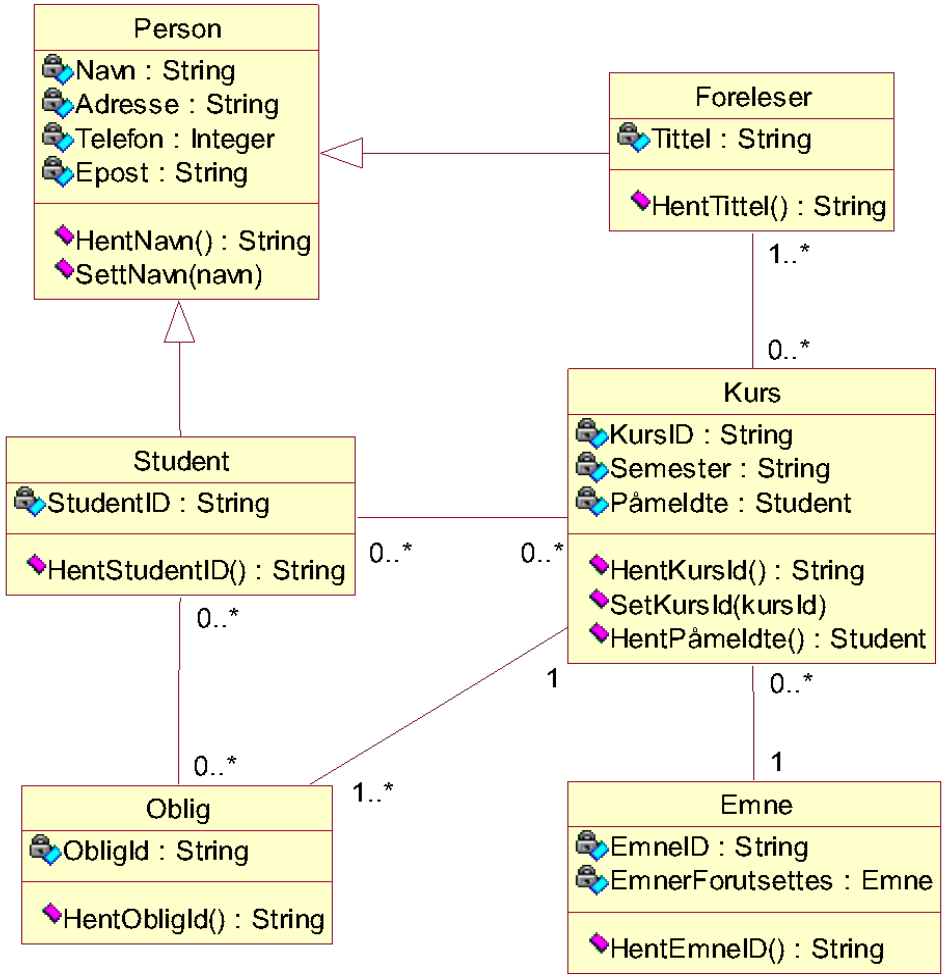
Sekvensdiagram “Se pasientinfo”



Generalisering

- Det er ofte nyttig å undersøke klassene i et system for å se om det er mulighet for generalisering
- I objektorienterte språk, som Java, er generalisering en del av språket – gjennom såkalt arvemekanisme (”inheritance”)
- Attributter og operasjoner (metoder) som er assosiert med ”superklasser” er også assosiert med ”subklasser” gjennom arv. Subklassene vil så legge til mer spesifikke attributter og operasjoner

Klassediagram - Student tar kurs



- Alle kurs har minst en obligatorisk oppgave.

Spesifikasjon av grensesnitt/interface

- Grensesnitt/Interface bør spesifiseres slik at objektene og andre komponenter kan designes i parallell.
- Ikke design representasjonen av data – kun “navn” og metoder (uten innhold). Innholdet defineres i objektene som “implementerer “grensesnittet.
- Objekter kan ha flere grensesnitt med ulike perspektiver av metodene som er spesifisert.
- Klassediagrammer blir brukt i UML for spesifikasjon av grensesnitt.

Eksempel (brukt i INF1010)
Ulike klasser samme grensesnitt (Interface)

```
interface Skattbar{                    // Skatt på importerte varer
        int skatt();
}

class Bil implements Skattbar{  // Bil: 100% skatt
       private String regNr;
       private int importpris;
       Bil (String reg, int imppris) {
              regNr = reg; importpris = imppris;
       }
       public int skatt( ){return importpris;}
       public String hentRegNr( ) {return regNr;}
}

class Ost  implements Skattbar{  // Ost: 200% skatt
       private int importprisPrKg;
       private int antKg;
       Ost (int kgPris, int mengde) {
              importprisPrKg = antKg; antKg = mengde;
       }
       public int skatt( ){return importprisPrKg*antKg*2;}
}
```

Objektdesign: Ansvarstilordning

- Ansvar er knyttet til objektet i form av dets oppførsel
  - *Handling:* Opprette objekt, beregne, initiere handlinger i andre objekter, kontrollere og koordinere handlinger i andre objekter.
  - *Kunnskap:* Vite om private data, relaterte objekter, ting som det kan utlede eller beregne
- Ansvar er ikke det samme som metoder, men metoder implementeres for å oppfylle ansvaret
- Kategorier av ansvar:
  - Sette (set) og hente (get) verdier av attributter
  - Opprette og initialisere nye instanser (objekter)
  - Hente fra og lagre til fil (ofte database)
  - Slette instanser
  - Legge til og slette linker for assosiasjoner (relasjoner)
  - Kopiere, konvertere og endre
  - Beregne numeriske resultater
  - Navigere og søke
  - …

Kjennetegn på ’god’ design

- En god utforming gjør den jobben den er ment å gjøre
- En god utforming er enkel og elegant

  Eleganse innebærer å finne akkurat riktig abstraksjonsnivå
- En god utforming er gjenbrukbar, utvidbar og enkel å forstå
- Et godt objekt har et lite og veldefinert ansvarsområde
- Et godt objekt skjuler implementasjonsdetaljer fra andre objekter

- Grady Booch

Analyse- vs. designmodell

- *Analysemodellen* utelater mange klasser som er nødvendige i et komplett system
  - Er typisk en domenemodell
  - Kan inneholde mindre enn halvparten av klassene i systemet.
  - Uavhengig av spesielle
    - brukergrensesnittsklasser
    - arkitekturklasser (f.eks. design patterns klasser)
- Den komplette *designmodellen* inneholder
  - Domenemodellen
  - Brukergrensesnittsklasser
  - Arkitekturklasser (f.eks. slik at klasser kan kommunisere)
  - Utility klasser (f.eks. håndtering av mengder og strenger)

Designprosesser

- Det finnes flere ulike objektorienterte designprosesser avhengig av organisasjonen som bruker prosessen
- Felles aktiviteter i disse prosessene inkluderer
  - Forstå og definere systemets kontekst og eksterne interaksjoner med systemet
  - Designe systemarkitekturen
  - Identifisere nøkkelobjektene i systemet
  - Utvikle designmodeller
  - Spesifisere grensesnitt

Designmønstre – ”Design patterns”

- Et designmønster er en måte å gjenbruke abstrakt kunnskap om et problem og løsningen på problemet
- Et mønster er en beskrivelse av et problem og essensen av løsningen
- Bør være tilstrekkelig abstrakt til å kunne bli gjenbrukt i ulike situasjoner
- I beskrivelser av mønstre brukes som regel objektorienterte teknikker som arv og polymorfisme (”Virtual methods”)

Hvorfor bruke mønstre (patterns) for informasjonssystemer?

- Mye av vårt daglige virke består i å lete etter strukturer (mønstre) i omgivelsene
- Mønstre er vanlige løsninger på vanlige problemer
- Det samme gjelder utvikling av informasjonssystemer
- Mange systemer har grunnleggende fellestrekk
- Det finnes mange mønstre (patterns) som er blitt utviklet gjennom systemutviklingens historie

Mer om Mønstre (”patterns”)

- Mønstre er navngitte retningslinjer for hvordan ansvar skal fordeles i ulike situasjoner.
- Mønstre brukes bl.a. i prosessen med å forfine sekvensdiagrammer
- GRASP – ’Patterns of General Principles in Assigning Responsibilites’ = Mønster for problem/løsning
- Sentrale prinsipper er
  - Ekspertprinsippet:
    - La det objektet som har kunnskapen (dataene) også behandle den
- Kontrollobjektprinsippet:
  - To typer kontrollere:
    - Fasadekontroller: En kontrollklasse har ansvar for alt (brukes i et lite system)
    - Use case kontroller: Styrer ett use case (brukes i større systemer. Ett kontrollobjekt for hvert use case).
- Skaperprinsippet:
  - Legg ansvar for å opprette et nytt objekt i klassen som må vite om det nye objektet

Ekspertprinsippet: (Information Expert)

- **Problem**: Hva er det generelle prinsipp for å tilordne ansvar til objekter?
- **Løsning**: La det objektet som har kunnskapen (dataene) også behandle den
- **Hvordan**:
  - Begynn med å formulere ansvarsområdet:
  - Eks: Student-kurs:

    Hvilket objekt har ansvar for å vite om hvilke emner som kreves for å ta et gitt emne?

    Hvilket objekt har ansvar for å gi en liste over alle studentene på et kurs?

Skaperprinsippet (Creator)

- Problem: Hvem er ansvarlig for å opprette nye objekter?
- Løsning: La det objektet som må vite om de nye objektene, lage dem
- Hvordan: Gi klasse B ansvaret for å opprette et objekt av klasse A dersom ett av følgende er sant:
  - B inneholder A-objekter
  - B registrerer A-objekter
  - B bruker A-objekter
  - B har data som sendes til A-objektet når det opprettes

Kontrollobjektprinsippet (Controller)

- Hvilken klasse skal behandle en systemhendelse/melding?
  - Kontrolleren ligger gjerne på klienten
  - Kontrolleren har bare metoder, få eller ingen attributter
  - Kontrolleren gjør ikke jobben selv, men mottar og fordeler oppgaver – er en slags administrator
  - Delegerer oppgaver og styrer use case
  - Er et bindeledd mellom brukergrensesnittet og applikasjonslaget (modellen)

Høy kohesjon

- Kohesjon er et mål på hva slags ansvar et objekt har og hvor fokusert ansvaret er
- Et objekt som har moderat ansvar og utfører et begrenset antall oppgaver innenfor ett funksjonelt område har høy kohesjon
- Objekter med lav kohesjon har ansvar for mange oppgaver innen ulike funksjonelle områder

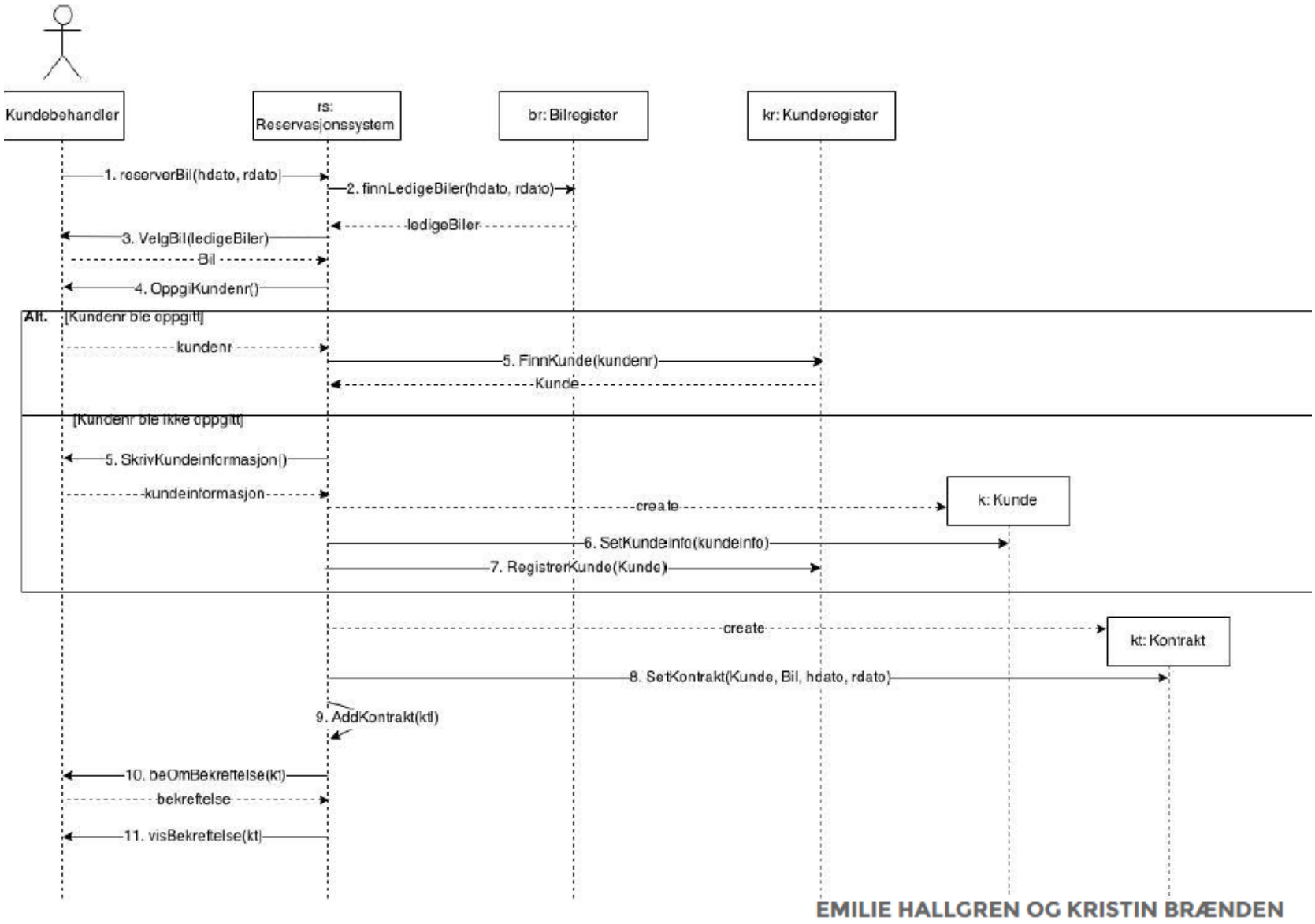
Lav kobling

- Kobling er et mål på hvor sterkt et objekt er knyttet til andre objekter
- Et objekt med sterk kobling er avhengig av mange andre objekter, noe som kan gjøre endring vanskelig

Analyse- vs. designmodell

- *Analysemodellen* utelater mange klasser som er nødvendige i et komplett system
  - Er typisk en domenemodell
  - Kan inneholde mindre enn halvparten av klassene i systemet.
  - Uavhengig av spesielle
    - brukergrensesnittsklasser
    - arkitekturklasser (f.eks. design patterns klasser)
- Den komplette *designmodellen* inneholder
  - Domenemodellen
  - Brukergrensesnittsklasser
  - Arkitekturklasser (f.eks. slik at klasser kan kommunisere)
  - Utility klasser (f.eks. håndtering av mengder og strenger)

Sekvensdiagram - Reserver bil



EMILIE HALLGREN OG KRISTIN BRÆNDEN

Pseudo-kode – Reserver bil

```
class Reservasjonssystem {
        // Disse objektene kjenner vi til fra før.
        Bilregister br;
        Kunderegister kr;
        Kundebehandler kb;
        //kb er her et objekt av klassen Kundebehandler – ikke med i sekvensdiagrammet

        ArrayList <Bil> ledigeBiler;
        Bil bil;
        String kundenr;
        Kunde k;
        Kontrakt kt;

        // metode som legger til en kontrakt i en liste
        AddKontrakt(kt) { }

        // kundebehandler velger tidsintervall (hentedato og returdato).
        reserverBil (hdato, rdato) {
                // Systemet returnerer en liste over tilgjengelige biler innenfor de spesifiserte
                datoene.
                ledigeBiler = bilregister.finnLedigeBiler(henteDato, returDato);
                // Kundebehandler velger én av bilene.
                bil = kb.velgBil(ledigeBiler);
```

Pseudo-kode – Reserver bil forts.

```
                // Systemet ber om kundenr ...
                kundenr = kb.oppgiKundenummer();
                if (kundenr) {
                        // … og finner kunden i systemet.
                        k = kr.finnKunde(kundenummer);
                // Alternativ flyt: Kunden finnes ikke.
                } else {
                        // Systemet oppretter ny kunde og fortsetter til neste steg
                        kundeinformasjon = kr.skrivKundeinformasjon();
                        k = new Kunde();
                        k.SetKundeInfo(kundeinfo)
                        kr.registrerKunde(k);
                }
                // Vi lager en ny kontrakt.'
                // Bil blir reservert i metoden nyKontrakt.
                kt = new Kontrakt();
                kt.SetKontrakt(k,bil,hdato,rdato);
                AddKontrakt(kt);
                // Systemet bekrefter at bilen er reservert for den gitte perioden.
                kb.visBekreftelse(kt);
        }
}
```

Oppgave - modellering

Du skal modellere deler av et system for kjøp av billetter til en IT konferanse for utviklere som varer i 3 dager. Det skal være mulig å kjøpe billetter for alle dager. Det er også mulig å kjøpe én eller to dagers billett med valg av dag(er).

Systemet har en betalingsmodul. Når betalingen er godkjent, blir billetten(e) tilgjengelige i PDF-format med en strekkode som skannes ved inngangen til konferansen. Det er også mulig i systemet å melde seg som frivillig hjelper for én eller flere dager. Man legger da inn en del opplysninger om seg selv og spesifiserer hva man helst ønsker å hjelpe til med (vakt, salg av mat/drikke, rydde etc.). Alle som blir godkjent som hjelpere, får gratis inngang for den eller de dagene man arbeider som frivillig.

Systemet har også en administrasjonsmodul som er tilgjengelig for ledelsen av konferansen. Denne modulen gir oversikt over antall personer som har kjøpt billetter til de ulike dagene, og den gir også oversikt over alle som har meldt seg som hjelpere til de ulike dagene. Noen med spesielle rettigheter i systemet kan godkjenne hvem som får være hjelpere.

a) Tegn et use-case diagram for systemet (få med alle aktører, og angi hvilke aktører som er primære og hvilke som er sekundære).
b) Tegn et sekvensdiagram for kjøp av billett til én eller flere dager. **Merk**: sekvensdiagrammet skal ikke modellere ønske om å være frivillig hjelper.
c) Tegn et klassediagram for systemet som også tar med muligheten for å bli hjelper.