

**IN2000**

# **Software Engineering med prosjektarbeid**

*Kravhåndtering og modellering.  
Objektorienterte prinsipper*

Yngve Lindsjørn og Steffen Almås

# I dag

- Kravhåndtering
  - hva og hvorfor
- Diagrammer
  - Use case, sekvens, aktivitet og klasse - *fra IN1030*
  - Tilstandsdiagram - *nytt diagram i IN2000* 🌟
- Design
  - Prinsipper og patterns

# Krav og kravhåndtering

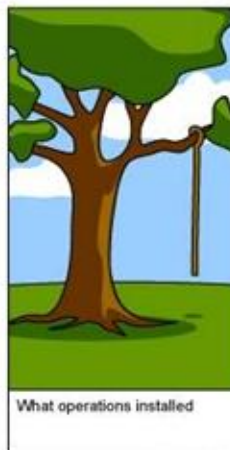
- Krav beskriver hva et system skal gjøre og hvilke begrensninger det må møte
- **Funksjonelle krav:**
  - Beskriver hva systemet skal gjøre, hvilke funksjoner det skal ha
  - Spesifiserer konkrete interaksjoner mellom brukere og systemet
  - Formuleres gjerne som **use case**
- **Ikke-funksjonelle krav:**
  - Beskriver systemets egenskaper og ytelse
  - Fokus på kvalitet og brukeropplevelser

# Kravinnsamlingsprosessen

1. **Innhenting:** Identifisere krav gjennom f.eks intervjuer, workshops, og analyser av eksisterende systemer.
2. **Dokumentasjon:** Kravene skrives ned i et strukturert format så de er klare og forståelige, f.eks som brukerhistorier eller use case.
3. **Validering:** Sikre at kravene er korrekte, komplette, og ikke i konflikt med hverandre.
4. **Prioritering:** basert på viktighet og risiko. Man bør fokusere på de mest kritiske kravene først.
5. **Endringskontroll:** Krav kan endres underveis. Endringer bør trackes på en systematisk måte.

# Utfordringer i kravhåndtering

- Uklare krav
- Endringer i krav
- Forskjellige interessenter med ulike behov og forståelse
- For dårlig validering
- Prioriteringer



# Gode beskrivelser av krav

er viktig for

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



# Ulike måter å beskrive krav

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

User stories   og use case 

# User stories - brukerhistorier

- I smidige metoder omformuleres krav til brukerhistorier
  - *Som bruker ønsker jeg å ...*
  - *Som utvikler ønsker jeg å ...*
- Beskriver et behov og hvorfor en funksjon er nødvendig
- Fokus på **brukeren/aktøren**

Som <bruker>  
ønsker jeg å <mål>  
for å <effekt>

Akseptansekriterier:

- ...

- ...

# Brukerhistorier - eksempel

*Som en **bruker av Vær-appen**  
ønsker jeg å se når det skal regne der jeg er  
for å vite om jeg trenger paraply 🌂*

Som <bruker>  
ønsker jeg å <mål>  
for å <effekt>

Akseptansekriterier:

- ...  
- ...

Akseptansekriterier (når brukeren er fornøyd med produktet):

1. Appen viser lokasjon (feks kart eller stedssøk i liste)
2. Brukeren kan se været et gitt tidsrom frem i tid

Det kan være vanskelig å skrive brukerhistorier med rett detaljeringsnivå!

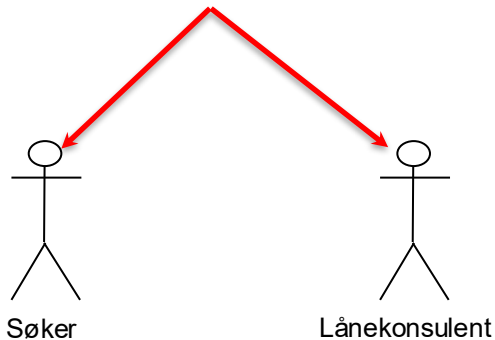
# Use case - brukstilfelle

- Beskriver hvordan aktører samhandler med systemet for å oppnå et mål
- Viser systemets funksjonalitet og samspillet mellom systemet og omgivelsene (brukere, andre systemer, komponenter)
- En mer formell og strukturert beskrivelse
- Høy detaljeringsgrad

# Use case modellering - finne aktører - lånesøknad

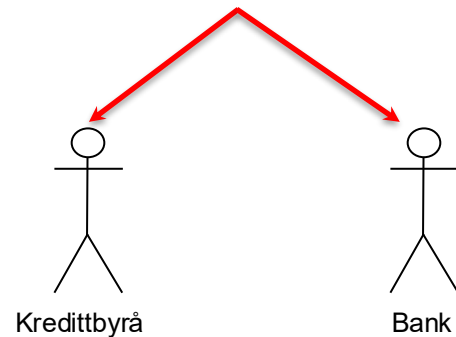
**Primære aktører** har egne mål, dvs. de initierer use case (en eller flere) som oppfyller deres mål.

## Primære aktører

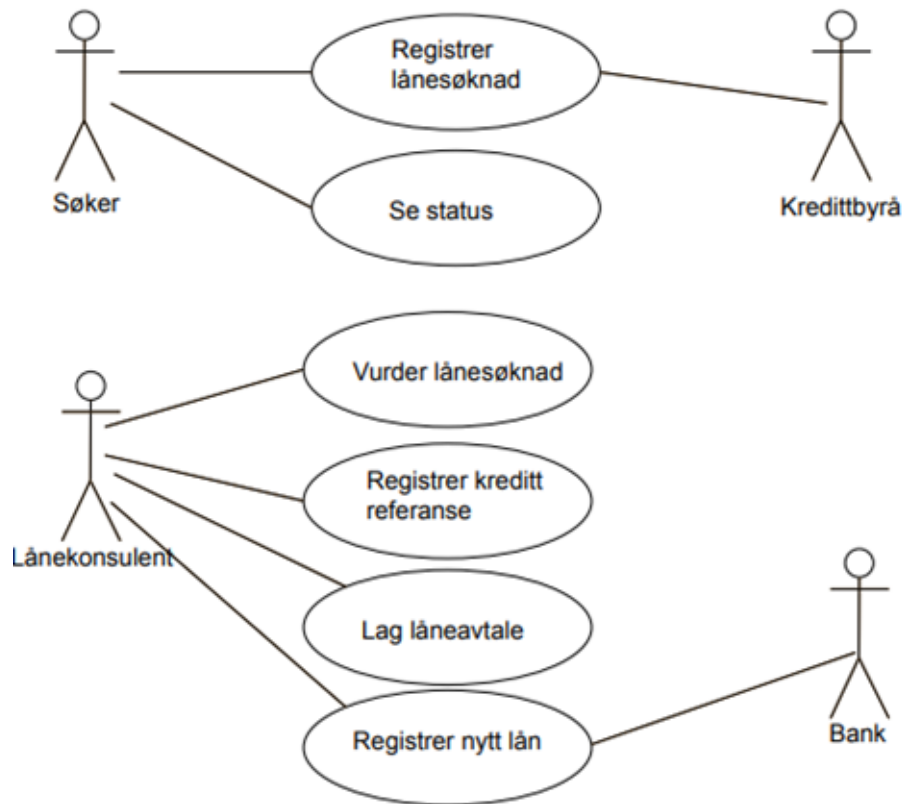


**Sekundære aktører** har ikke egne mål, men er nødvendige for å realisere målene til de primære aktørene

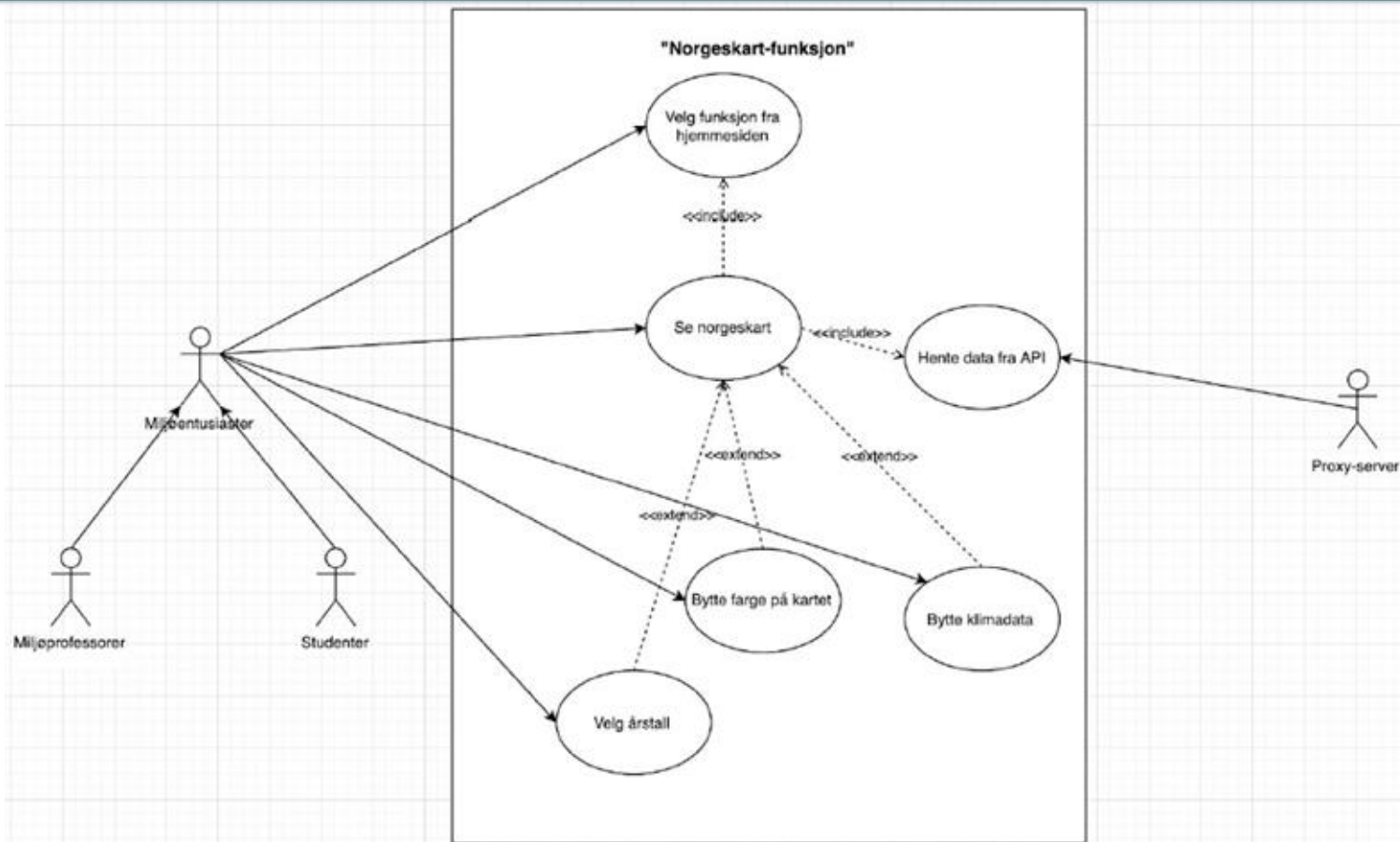
## Sekundære aktører



# Use case diagram - Eksempel: Lånesøknad



# Eksempel USE-case fra IN2000 - klima case

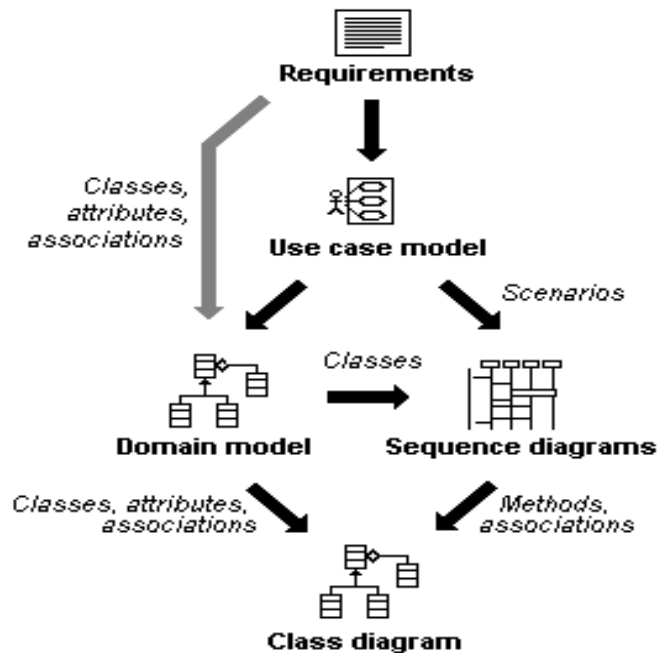


# Use case i prosjektplanlegging

- Planlegg hvilke use case som skal implementeres i hvilke iterasjoner av prosjektet:
  - Prioriter use casene etter hvor viktige de er og/eller hvor vanskelige de antas å være å implementere.
  - Hovedflyten implementeres først, deretter alternativene.
  - Estimer hvor mange use case (eller hendelsesflyt og alternativ flyt) som kan implementeres i en iterasjon.
  - Dette kommer an på en rekke faktorer, som lengde på iterasjon, antall timer og ressurser dedikert til oppgaven m.m.

# Use case i design

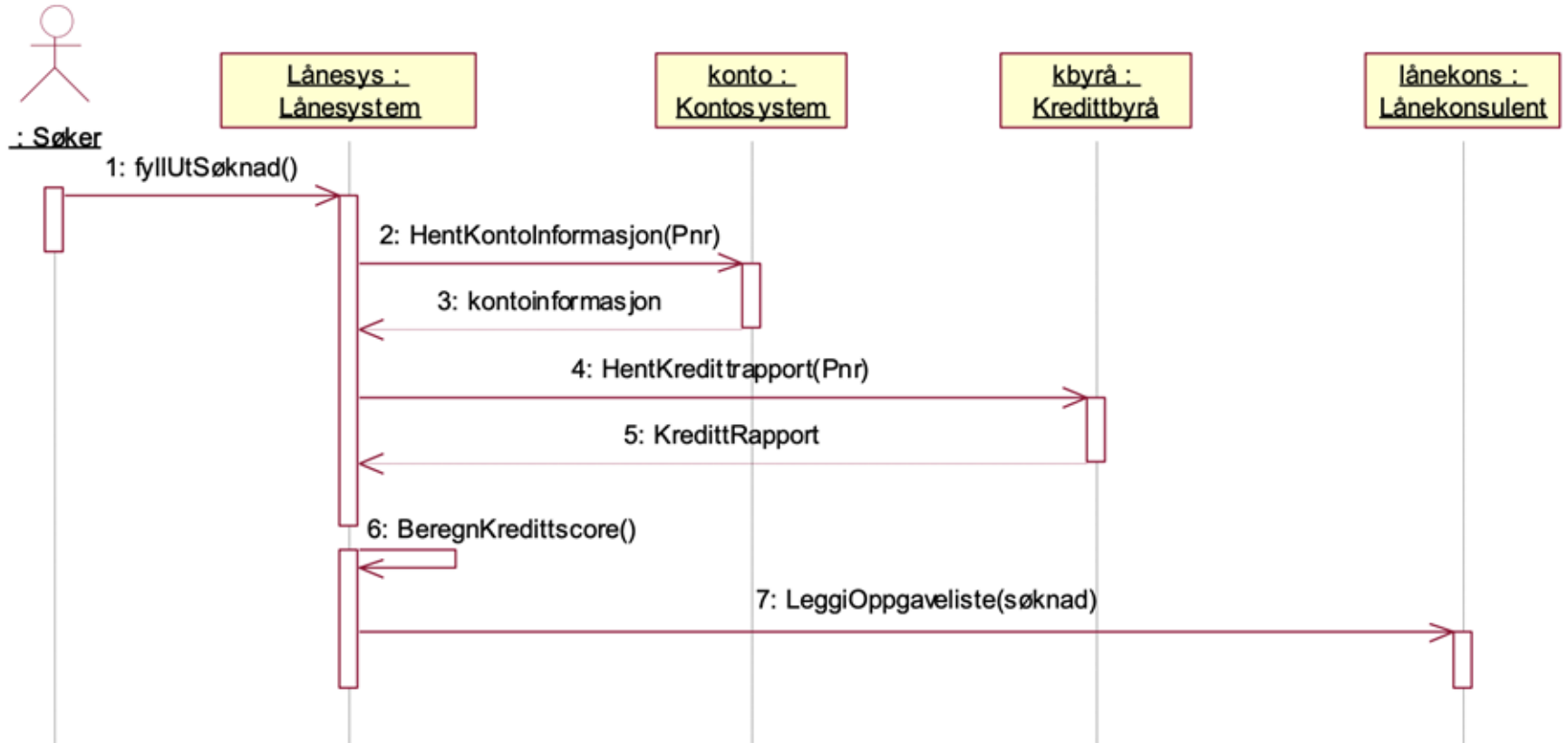
- Krav danner grunnlag for use case-modeller
- Hendelsesflyt i use casene detaljeres ut i **sekvensdiagram**
- Domenemodellen utvides til **klassediagram** med systemklasser



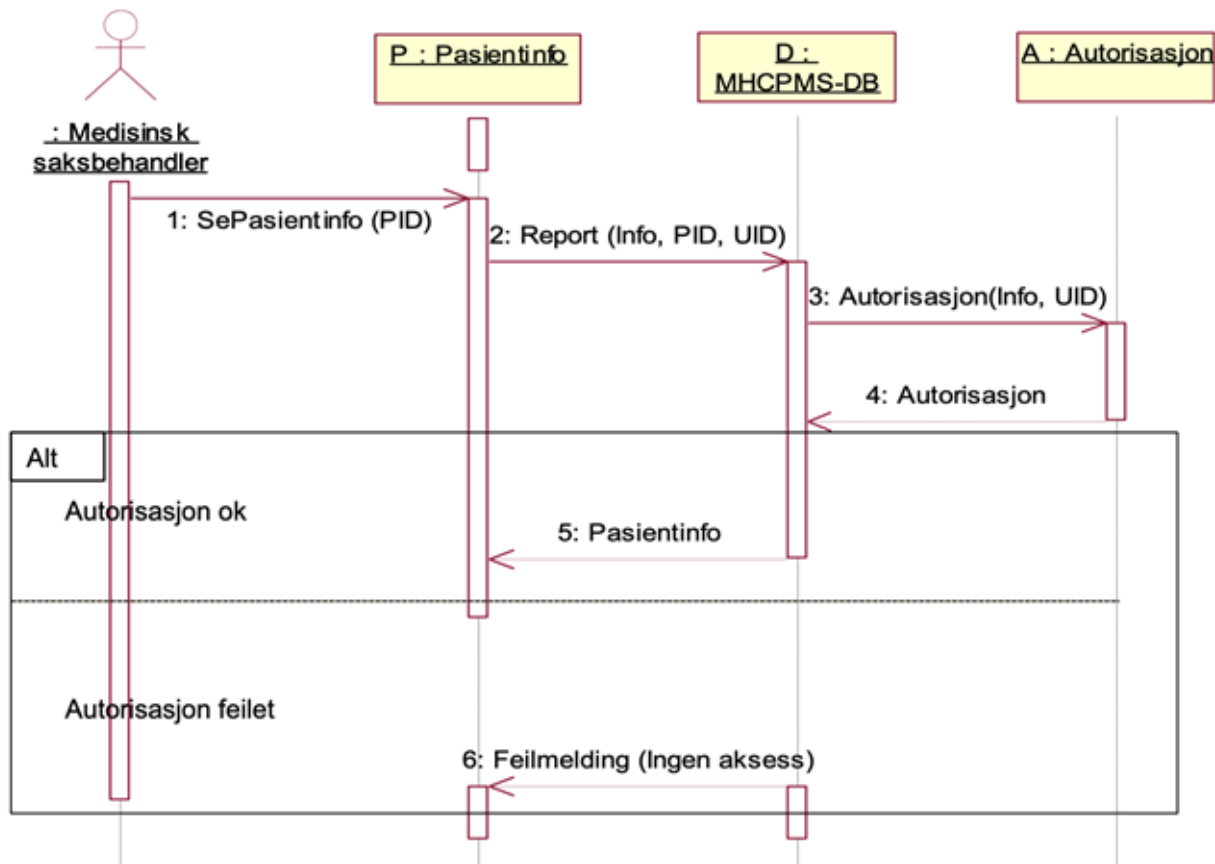
# Sekvensdiagram

- 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) i et use case vises som meldinger som sendes mellom objektene ved kall på objektenes metoder.

# Sekvensdiagram - Eksempel: Lånesøknad



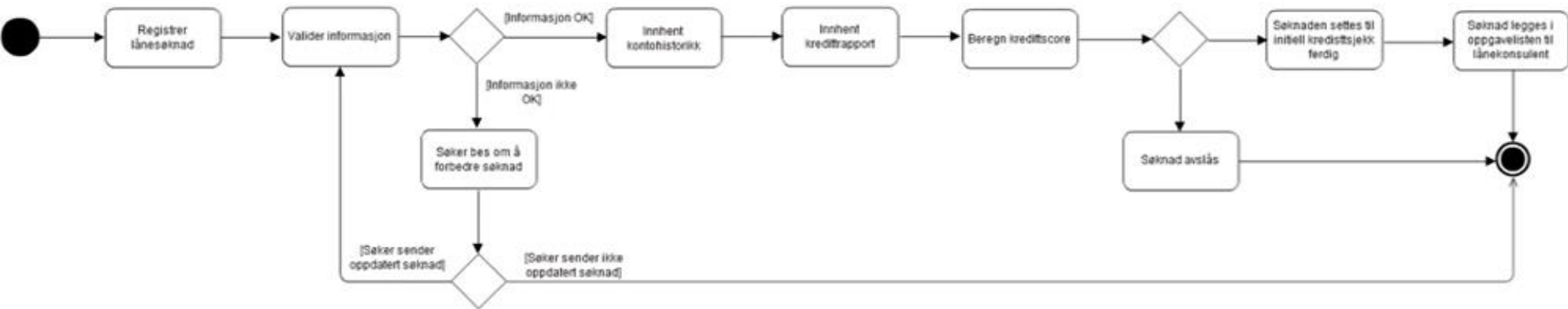
# Sekvensdiagram - Eksempel: “Se pasientinfo”



# Aktivitetsdiagram

- Et aktivitetsdiagram kan grafisk representere hendelsesflyten i et use case
- Stegene i use casene vises som aktiviteter
- Beslutninger underveis vises som “” (diamant)
- Aktivitetsdiagrammer og sekvensdiagrammer brukes noe overlappende, men **sekvensdiagrammer er typisk mer kodenært** mens **aktivitetsdiagrammer er mer forretningsnært**.

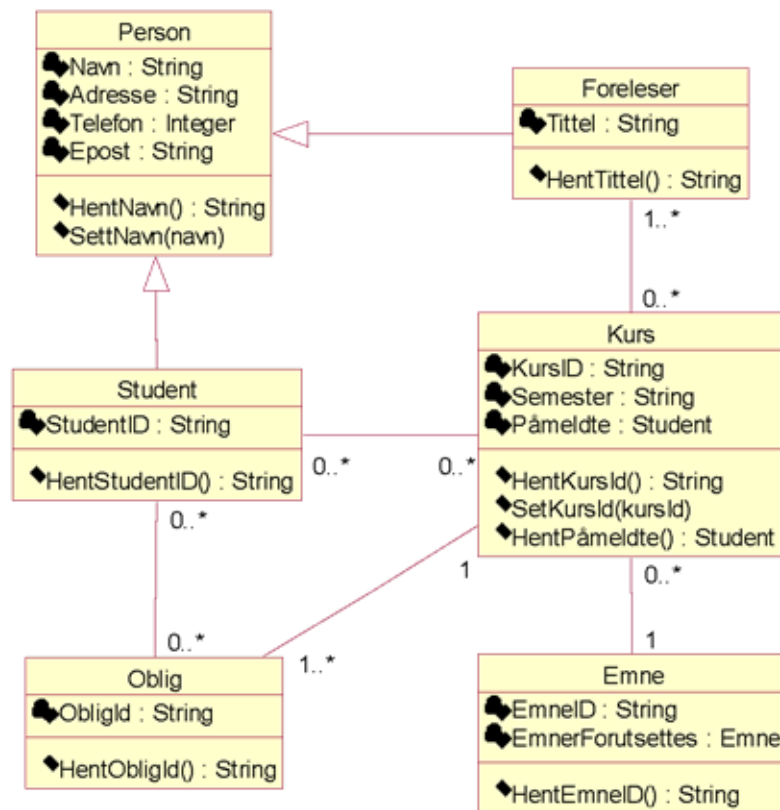
# Aktivitetsdiagram - Eksempel: Registrere lånesøknad



# Generalisering

- Det er ofte nyttig å undersøke klassene i et system for å se om det er mulighet for generalisering.
- I objektorienterte språk 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 - Eksempel: Student tar kurs



\*Multiplisitet angir at det er minst en obligatorisk oppgave per kurs

# Spesifikasjon av grensesnitt/interface

- Grensesnitt/Interface bør spesifiseres slik at objektene og andre komponenter kan designes parallelt.
- 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.

# Eksempel fra IN1010

```
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;}
}
```

# Hendelsesdrevet (“Event-driven”) modellering

- Mange systemer er drevet av ”hendelser”, med minimal data prosessering
- Modellene viser hvordan et system reagerer på eksterne og interne hendelser
- Baseres på antagelsen av at systemet har et endelig antall tilstander og at hendelser (stimuli) fører til at systemet går fra en tilstand til en annen

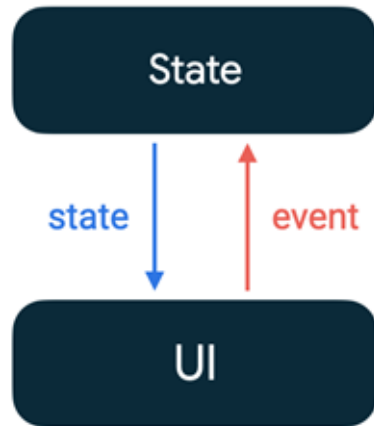
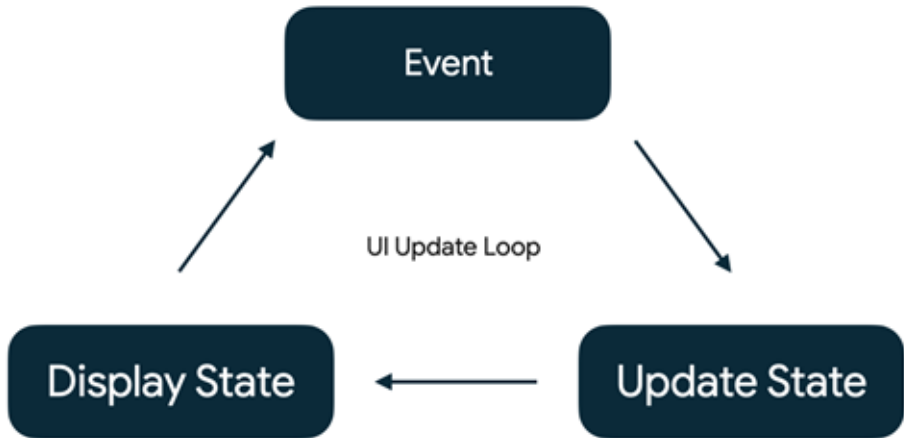
# Android - Tilstand (repetisjon)

- En app sin “tilstand” inneholder verdier som kan endres over tid. Dette kan være alt fra en database til en variabel i en klasse
- Noen eksempler er:
  - Hva telleren er på nå
  - Hvor langt det er scrollet i en liste
  - Meldinger en har mottatt
  - Om noe er “expanded” eller ikke
- Compose har et eget system for å håndtere endringer i tilstand

# Android - UI update loop (repetisjon)

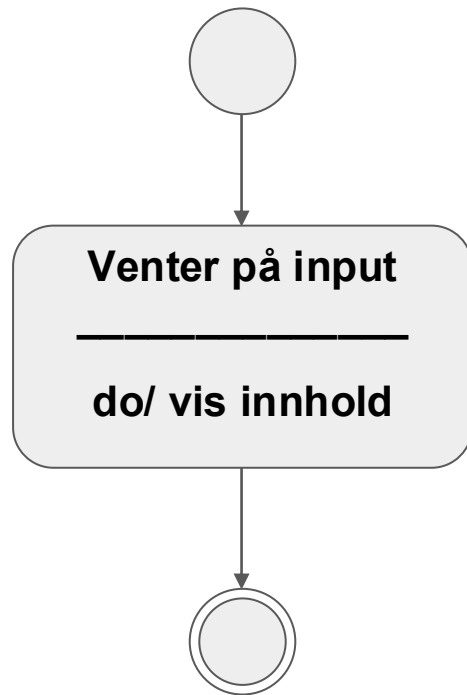
State definerer hva som vises i brukergrensesnittet til enhver tid

- Tilstand *er*
- Hendelser *skjer*



# Tilstandsdiagram - Om

- Visualiserer tilstander og hendelser til et system
- Ganske lik syntaks som aktivitetesdiagram
  - Samme start/slutt-symbol
- **Tilstander** modelleres som bokser
  - **do/** - skjer fortløpende/gjentatte ganger i gitt tilstand
  - **exit/** - skjer kun når tilstanden endres/er over
- **Tilstandsendring** modelleres som piler

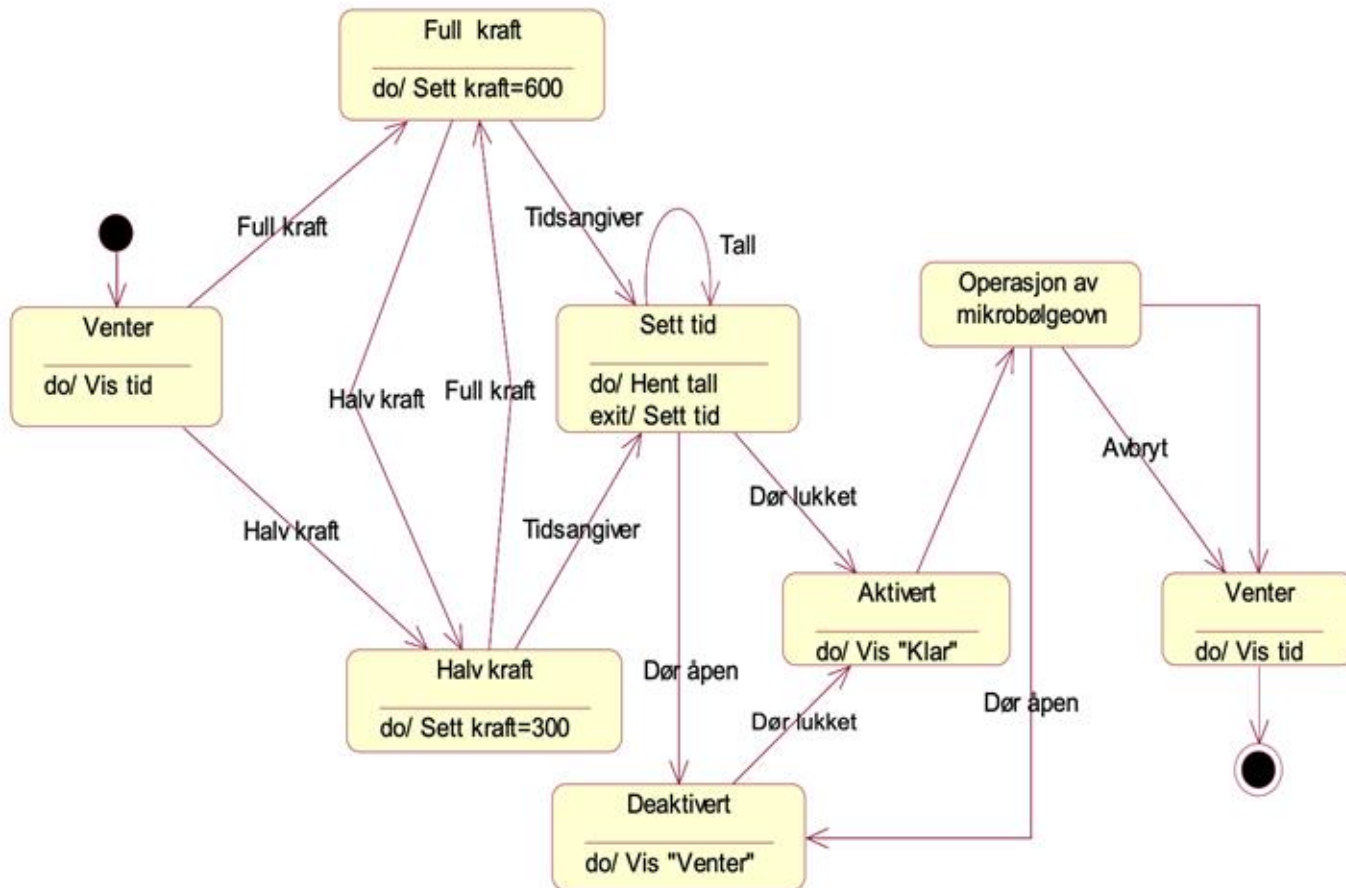


# Tilstandsdiagram - Eksempel: Mikrobølgeovn

Tilstander og hendelser for en tenkt mikrobølgeovn

|                    |                                                                |
|--------------------|----------------------------------------------------------------|
| Halv kraft (300 W) | Brukeren har trykket halv kraft knappen                        |
| Full kraft (600 W) | Brukeren har trykket full kraft knappen                        |
| Tidsinnstilling    | Brukeren har trykket en av de forhåndsgitte tidsinnstillingene |
| Tall               | Brukeren har trykket et tall på tastaturet                     |
| Åpen dør           | Døra er ikke lukket                                            |
| Dør lukket         | Døra er lukket                                                 |
| Start              | Brukeren har trykket startknappen                              |
| Kanseller          | Brukeren har trykket kanseller knappen                         |

# Tilstandsdiagram - Mikrobølgeovn



# 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

# Kvalitetsattributter (se også lærebok tabell 4.2 og kapitel 8)

- **Respons** – Er responstiden god nok?
- **Pålitelighet** – Virker funksjonaliteten som forventet?
- **Tilgjengelighet** – Leveres funksjonaliteten når brukerne ønsker det?
- **Sikkerhet** – Er systemet sikkert nok når det gjelder uautorisert inngrep eller angrep på systemet?
- **Brukskvalitet** – Kan brukerne navigere i systemet på en rask og feilfri måte?
- **Vedlikeholdbarhet** – Kan systemet videreutvikles og oppdateres med ny funksjonalitet uten for store kostnader?
- **Robusthet** – Kan systemet levere tjenester etter en større feil eller angrep?

## Tips: Fra kapittel 8 i lærebok

- Bruk en programmeringsstil som reduserer sjansen for å introdusere feil
- Lag kode som er forståelig og lett å lese
- Unngå språkkonstruksjoner som lett blir «utsatt for feil»
- Valider input
  - Alder er et tall etc.
- Refaktorer koden – fjern uklar og uforståelig kode
- Gjenbruk velprøvd og sikker kode, og bruk velprøvde og konstruksjoner som er testet, slik som design patterns.

# Code smells (Figure 8.2 i lærebok)

**Problem** – forstår ikke problemet eller problemområdet (domene)

**Teknologi** – Programmeringsspråk, bibliotek, IDE, database

**Kompleks program** - mange komponenter som virker sammen – kan være vanskelig å forstå hvordan tilstanden til programmet endres

# Code smell - karakteristikk i kildekoden som kan indikere et større problem

- **Store klasser** – del opp i mindre klasser som er lettere å forstå (prinsippet om ett ansvarsområde – høy kohesjon)
- **Store/lange metoder/funksjoner** – Splitt lange metoder i mindre, mer spesifikke metoder og funksjoner. Gjør kun en ting.
- **Duplisert kode** – Må endre flere steder. Lag en singel instans av den dupliserte koden.
- **Meningsløse navn på variable etc.** – tyder ofte på hastverk. Erstatt med mer meningsfulle navn
- **Ubrukt kode** – reduserer lesbarheten av koden. Fjern denne koden

# Høy kohesjon og lav kobling

## 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 et 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.

Pause



# Design patterns

# 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
- Fra lærebok (kap 8.1.2): «A general reusable solution to a commonly occurring problem within a given context in software design»

# Mer om Mønstre ("patterns")

- Mønstre er navngitte retningslinjer for hvordan ansvar skal fordeles i situasjoner.
- Mønstre brukes bl.a. i prosessen med å forfine sekvensdiagrammer.
- GRASP – 'Patterns of General Principles in Assigning Responsibilities' = Mønstre for problem/løsning
- Sentrale prinsipper er
  - Ekspertprinsippet
  - Kontrollobjektprinsippet
  - Skaperprinsippet

# Ekspertprinsippet (Information Expert)

- **Problem:** Hva er det generelle prinsippet 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)

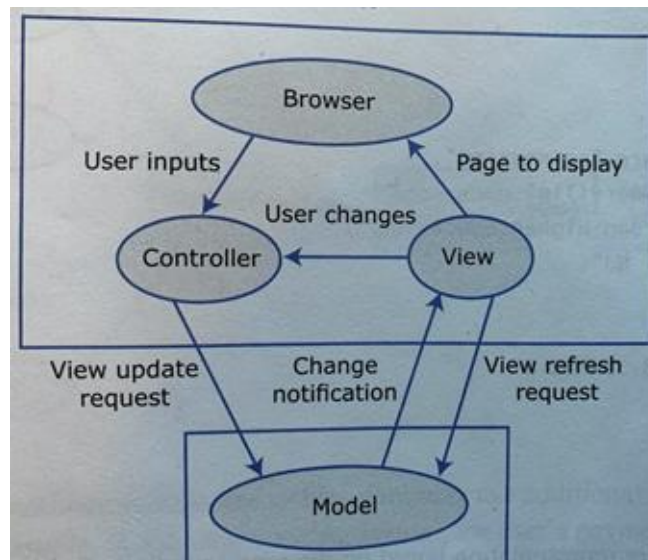
- **Problem:** Hvem er ansvarlig for å opprette nye objekter?
- **Løsning:** La det objektet som må vite om de nye objektene, lage dem
- **Hvordan:** Gi klasse B ansvaret for å opprette et objekt av klasse A dersom ett av følgende er sant:
  - B inneholder A-objekter
  - B registrerer A-objekter
  -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

# MVC

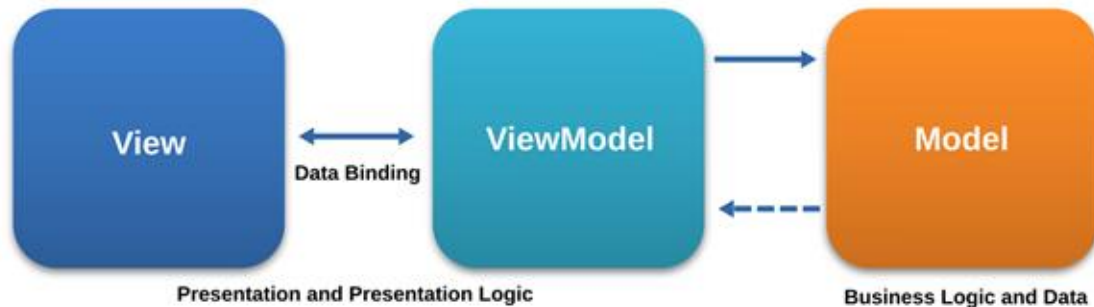
- “Klient – server” applikasjoner bruker ofte MVC pattern
- Klienten oppdateres når data endres på serveren
- «model» representerer system data og «business» logikk –
- Alle «views» blir oppdatert når modellen endres



Fra lærebok figur 4.13

# Model-View-ViewModel (MVVM)

- **Model:** Håndterer data og forretningslogikk. Kan også kalles “ryggraden” i appen din siden den har kontroll på hvordan ting skal beregnes og lagres.
- **View:** Utseende til appen - det brukeren ser og interagerer med.
- **ViewModel:** Fungerer som en “bro” mellom Model og View. Henter data fra Model og gjør den klar til å vises i View, i tillegg tar eventuelle input fra View og sender det tilbake til modellen



# Mermaid - Hva

- Språk for å lage diagrammer
- [Open Source-prosjekt](#)



- *“JavaScript based diagramming and charting tool that renders Markdown-inspired text definitions to create and modify diagrams dynamically.”*

# Mermaid - Hvorfor

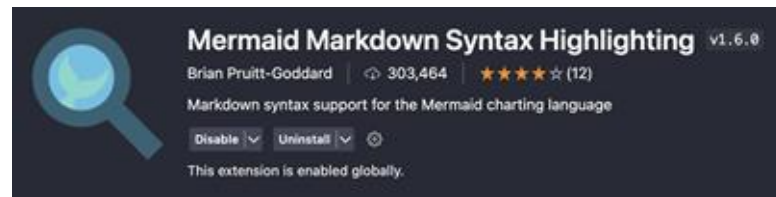
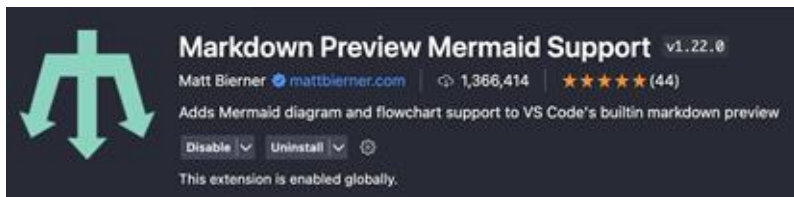
- “Diagrammer som kode”
- Diagrammer representerer ofte noe teknisk, mer nærhet til koden de representerer
- Felles måte å lage diagrammer på
- **Versjonsstyring med git**

# Mermaid - Diagramtyper

- [Aktivitetsdiagram](#)
- [Sekvensdiagram](#)
- [Klassediagram](#)
- [Tilstandsdiagram](#)
- [git graph](#)
- [Gantt](#)
- [Kakediagram](#)
- Og MANGE flere!

# Modellere Mermaid-diagrammer i IN2000

- Markdown plugin i Android Studio har en bug...
- Så dere kan velge mellom følgende løsninger:
  - [Mermaid Online Editor](#)
  - [VS Code](#) (husk plugins!)
    - Markdown Preview Mermaid Support
    - Mermaid Markdown Syntax Highlighting



# Demo

- VS Code
  - Sekvensdiagram
  - Aktivitetsdiagram
- Og litt git ;)

# Demo - Sekvensdiagram m/API

- **Oppgave:** Lag et Sekvensdiagram som viser at en bruker interagerer med en app som henter data fra API. Du skal modellere en alternativ flyt dersom API-et ikke responderer. Du kan anta at du har metoden *fetchWeatherData()* til disposisjon.

# Demo - Sekvensdiagram m/API

- **Oppgave:** Lag et Sekvensdiagram som viser at en **bruker** interagerer med en **app** som henter data fra **API**. Du skal modellere **en alternativ flyt** dersom API-et ikke responderer. Du kan anta at du har metoden ***fetchWeatherData()*** til disposisjon.
- Finn aktuelle “roller”:
  - Bruker (aktør)
  - App (system)
  - API (system)
- Annet:
  - En alternativ flyt
  - Metoden ***fetchWeatherData()*** skal brukes

**Demo**

# Demo - Aktivitetsdiagram - Pante flasker

- **Oppgave:** Lag et aktivitetsdiagram for en panteautomat basert på følgende tekstlige beskrivelse av et use case:
- Tekstlig beskrivelse for use caset “***Pant av flasker***”
  - **Navn:** Pante flasker
  - **Aktør:** Kunde
  - **Prebetingelse:** Panteautomat er klar til å ta imot pant
  - **Postbetingelse:** Kunde får ut kvittering eller lodd i pantelotteriet

# Demo - Aktivitetsdiagram - Finn aktiviteter

## 1. **Identifiser** de ulike aktivitetene som inngår i use case “Pant av flasker”

- Sett inn panteobjekt
- Ta ut panteobjekt
- Legg til pantebeløp
- Skriv ut lodd
- Skriv ut kvittering

Representeres som avrundede rektangler i aktivitetsdiagrammet

# Demo - Aktivitetsdiagram - Finn avgjørelser og utfall

**2. Identifiser** de ulike avgjørelsene og mulige utfall som inngår i use case  
“Pant av flasker”

- **Godkjent?**

- Er panteobjektet gyldig?

- **Fortsett?**

- Ønsker brukeren å pante flere objekter?

- **Kvittering eller lodd?**

- Ønsker brukeren kvittering eller lodd?

**Demo**