Dagens forelesning

- Litt mer om design med UML sekvensdiagrammer
  - Sentralisert og delegert kontrollstil
    - Resultater fra et eksperiment
- UML klassediagrammer
  - Notasjon: UML klassediagram og objektdiagram
  - Metode: Fra sekvensdiagram til klassediagram
  - Litt om persistens: Lagring av objekter i OO databaser og relasjonsdatabaser

Metode for ansvarsdrevet OO

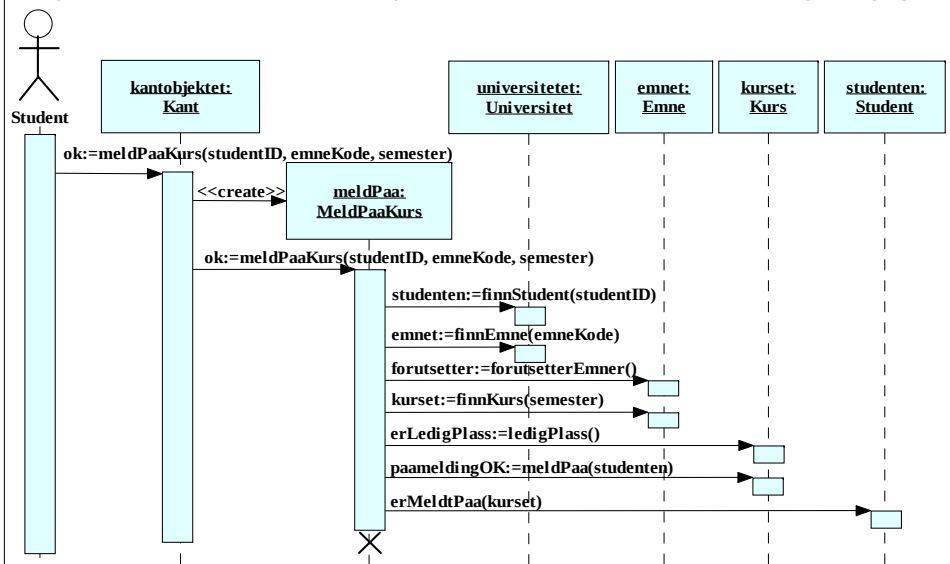
- Inf1050 metoden (Iterativ):
  - Analyse av krav
    - (1) Identifiser aktører og deres mål
    - (2) Lag et høynivå bruksmønsterdiagram
    - (3) Spesifiser hvert bruksmønster tekstlig med normal hendelsesflyt og variasjoner
  - Objektdesign
    - For hvert bruksmønster:
      - (4) Identifiser objekter og fordel ansvar mellom dem (CRC)
      - (5) Lag sekvensdiagram for normal hendelsesflyt og viktige variasjoner
      - (6) Lag klassediagram som tilsvarer sekvensdiagrammene
    - (7) Lag til slutt klassediagram på systemnivå

Delegering av ansvar i en trelagsarkitektur

- Forretningsobjekter (“entity objects”)
- Kontrollobjekter (”control objects”)
- Kantobjekter (“boundary objects”)
- Hvor mye ansvar bør kontrollobjektene ha, og i hvilken grad bør vi ”bevisstgjøre” forretningsobjektene våre??

Normal hendelsesflyt for ”Meld på kurs”

(sentralisert kontrollstil, kontrollobjektet har ansvar for det meste av handlingsforløpet)



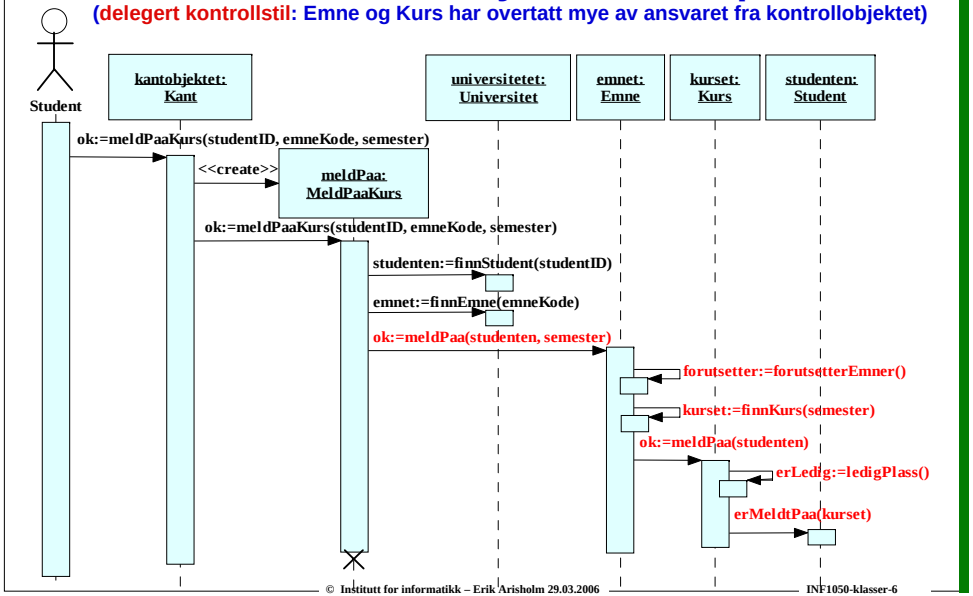
Normal hendelsesflyt for ”Meld på kurs”

(Eksempel på Java kode for metoden meldPaaKurs i en sentralisert kontrollstil)

```
public class MeldPaaKurs {
  ...
  public boolean meldPaaKurs(String studentID, String emneKode, String semester)
  {
    // antar at objektet ”universitetet” er tilgjengelig:
    Student studenten = universitetet.finnStudent(studentID);
    Emne emnet = universitetet.finnEmne(emneKode);
    boolean forutsetter = emnet.forutsetterEmner();
    Kurs kurset = emnet.finnKurs(semester);
    boolean erLedigPlass = kurset.ledigPlass();
    boolean paameldingOK = kurset.meldPaa(studenten);
    studenten.erMeldtPaa(kurset);
  }
}
```


Normal hendelsesflyt for ”Meld på kurs”

(delegert kontrollstil: Emne og Kurs har overtatt mye av ansvaret fra kontrollobjektet)

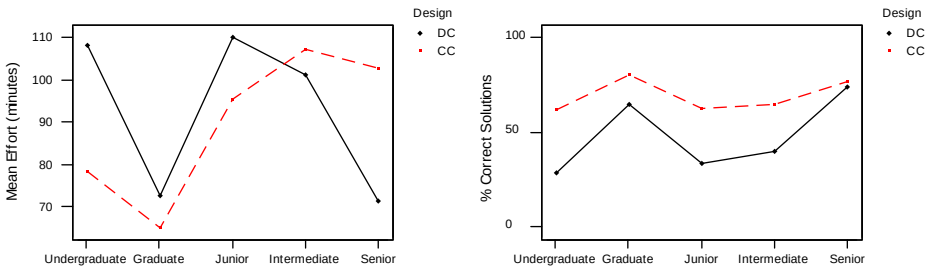


Normal hendelsesflyt for ”Meld på kurs”

(Eksempel på Java kode for metoden meldPaaKurs i en delegert kontrollstil)

```
public class MeldPaaKurs {
  ...
  public boolean meldPaaKurs(String studentID, String emneKode, String semester)
  {
    // antar at objektet "universitetet" er tilgjengelig:
    Student studenten = universitetet.finnStudent(studentID);   // message #1.2.1
    Emne emnet = universitetet.finnEmne(emneKode);              // message #1.2.2
    boolean ok = emnet.meldPaa(studenten, semester);            // message #1.2.3
  }
}
```


Resultater fra et kontrollert eksperiment (*)



DC = Delegated Control Style
CC = Centralized Control Style

Totalt 158 Java-utviklere deltok, og skulle gjøre endringer på enten et *DC* eller et *CC* design alternativ for det samme systemet.

Vi målte tid (”Mean Effort”) og kvalitet (”% correct solutions”)

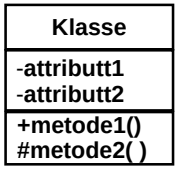
Kun seniorkonsulentene ser ut til å gjøre oppgavene bedre med et *DC* design

*** Erik Arisholm and Dag Sjøberg, ”Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software,” *IEEE Transactions on Software Engineering*, 2004**

Delegert vs sentralisert kontrollstil

- **Sentralisert kontrollstil:**
  - Lett å få oversikt over hva som skjer i et bruksmønster
  - Feilsituasjoner/variasjoner som krever tilbakemeldinger fra en aktør (via kantobjektene) kan enkelt håndteres av kontrollobjektet
  - Introduserer flere avhengigheter mellom kontrollobjekt og forretningsobjekter. Potensielt mindre gjenbrukbar/vedlikeholdbar kode
- **Delegert kontrollstil:**
  - Mer ”elegant” objektorientert design, men
  - Overdreven bruk av delegering gjør det vanskelig å få oversikt (spesielt dersom sekvensdiagrammer ikke er tilgjengelige!)
  - Litt mer komplisert å håndtere feilsituasjoner/variasjoner som krever tilbakemeldinger fra en aktør (siden all kommunikasjon med kantobjektene må gå via kontrollobjektene)

UML – Klasser og objekter



+ betyr ”public”

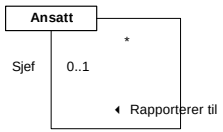
- betyr ”private”

 # betyr ”protected”

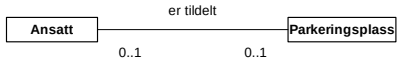
En klasse beskriver hva objektene vet (attributter) og hvilke meldinger de forstår (metoder). Objektene er forekomster av klassebeskrivelsen.

Assosiasjoner

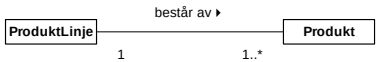
Refleksiv assosiasjon



Binær assosiasjon

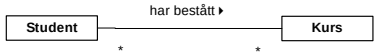


(en til en)



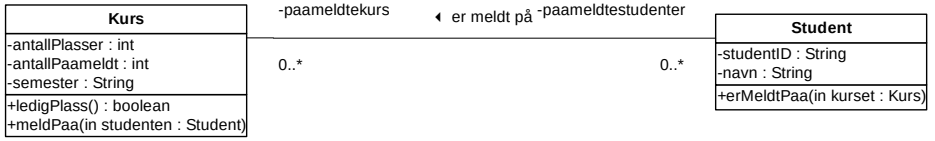
(en til mange)

NB! * = 0..*



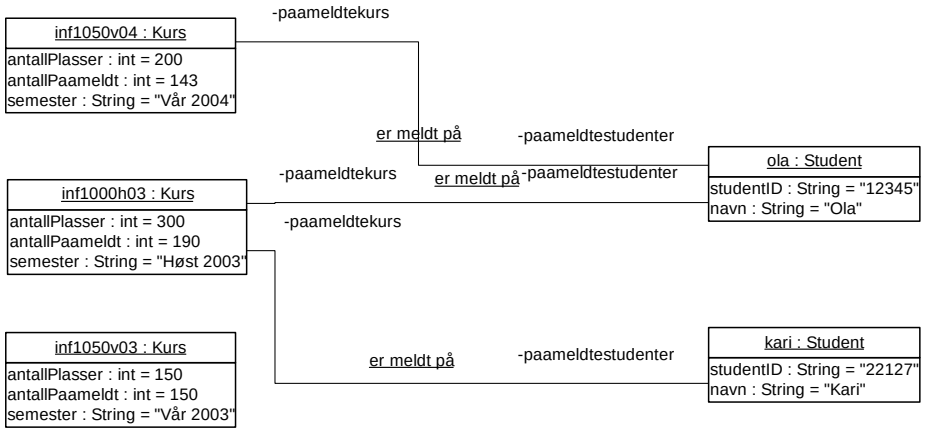
(mange til mange)

Eksempel på UML klassediagram



NB! Fremmednøkler osv vises ikke i et OO klassediagram

Eksempel UML objektdiagram



Infostoff: Eks. på realisering i Java

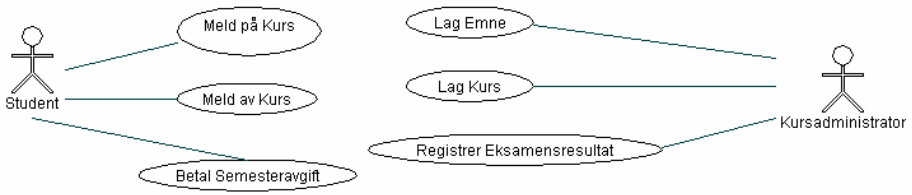

```
class Kurs {

   //assosiasjoner
   Emne emnet;                    // referanse til emnet for kurset
   Vector paameldtestudenter;     // liste over påmeldte studenter

   // attributter
   private int antallPlasser;
   private int antallPaameldt;
   private String semester;

   public boolean ledigPlass()   {}

   public boolean meldPaa(Student s)
   {
         antallPaameldt = antallPaameldt + 1;
         paameldtestudenter.addElement(s);
   }
}
```

```
class Student {

   //assosiasjoner
   Vector paameldtekurs;          // liste over påmeldte kurs

   // attributter
   private String studentID, navn;

   public boolean erMeldtPaa(Kurs k)
   {
         paameldtekurs.addElement(k);
   }
}
```


Sammenhengen mellom sekvensdiagram og klassediagram

- **Start med sekvensdiagrammet for normal hendelsesflyt:**
  - **Lag klasser for alle objektene.**
  - **Hver unike melding til et objekt resulterer i en metode for klassen til objektet.**
  - **Legg til de attributtene som metodene bruker**
  - **Lag nødvendige assosiasjoner for at klassene skal kunne utveksle meldinger mellom sine objekter**
- **For hvert sekvensdiagram for en variasjon:**
  - **Legg til nye klasser, attributter, metoder og assosiasjoner i klassediagrammet basert på meldingene i sekvensdiagrammet for variasjonen.**

Kursregistrering bruksmønstermodell



Endringer i modellen sammenlignet med tidligere forslag:
1) Vi skiller mellom "Emne" og "Kurs"
2) "Lag Emne" og "Lag Kurs" inneholder "Nytt Emne" og "Nytt Kurs" som variasjoner
3) Bruksmønster "Behandle dispensasjonssøknad" utgår
4) Variasjon på "Meld på kurs":
   Dersom et kurs er fullt kan studenten velge å bli satt på venteliste.
5) Variasjon på "Meld av kurs":
   Det finnes studenter på venteliste: Første på ventelista får plassen

Spesifikasjon av ”Meld på kurs”

Navn: Meld på kurs

Aktør: Student

Trigger: Student ønsker å melde seg på et kurs

Pre-betingelse: Student har betalt semesteravgift og er logget inn på systemet

Post-betingelse: Student er meldt på kurset eller er satt på venteliste

Normal Hendelsesflyt:

1. Studenten velger emne
2. Systemet sjekker at studenten kvalifiserer til å ta emnet
3. Systemet finner kurs for emnet
4. Systemet sjekker om det er ledig plass på kurset
5. Systemet registrerer studenten på kurset

”Meld på kurs” (forts.)

Variasjoner:

1a. Emnet finnes ikke:

1. Studenten velger et annet emne eller avslutter

2a. Emnet forutsetter andre emner:

1. Systemet sjekker at studenten har bestått kurs for emner som forutsettes

1a. Studenten har ikke bestått kurs for emner som forutsettes:

1. Studenten velger et annet emne eller avslutter

3a. Det holdes ikke kurs i emnet dette semesteret:

1. Studenten velger et annet emne eller avslutter

4a. Kurset er fullt:

1. Systemet spør om studenten ønsker å bli satt på venteliste

1a. Studenten ønsker å bli satt på venteliste:

1. Systemet setter studenten på venteliste

Relatert informasjon:

I denne versjonen holdes administrasjon av gruppeundervisning utenfor systemet

Eks: CRC-kort for bruksmønsteret ”Meld på kurs”

Kant <<boundary>>	MeldPaaKurs
Kommuniserer med aktøren og kontrollobjektet	

Universitet <<entity>>	Emne Student
Oppslagsobjekt som vet hvilke emner og studenter som finnes i systemet	

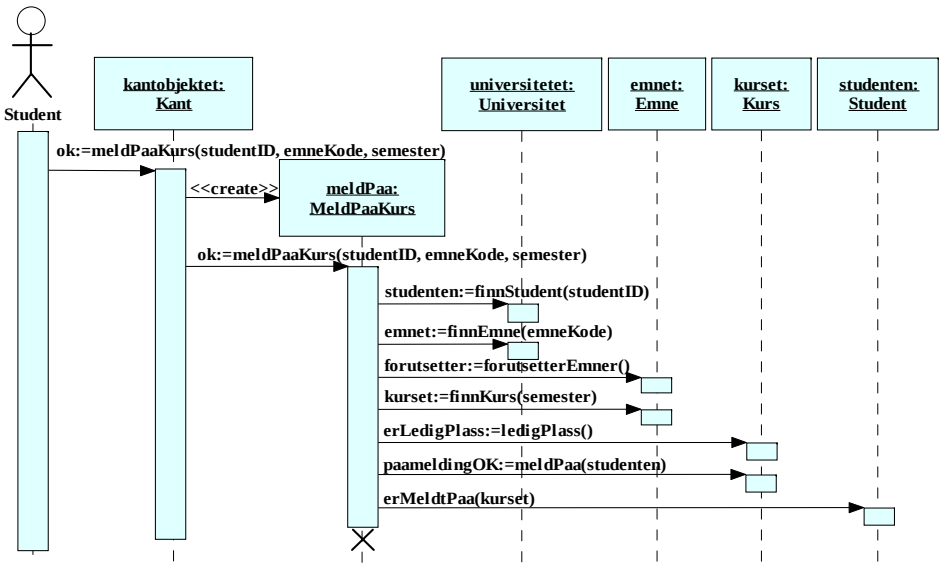
Emne <<entity>>	Kurs
Vet min emnekode Vet hvilke emner som forutsettes Vet hvordan man finner kurs i emnet	

MeldPaaKurs <<control>>	Universitet Emne Kurs Student Kant
Kontrollerer hendelses-forløpet i bruksmønsteret ”Meld på kurs”	

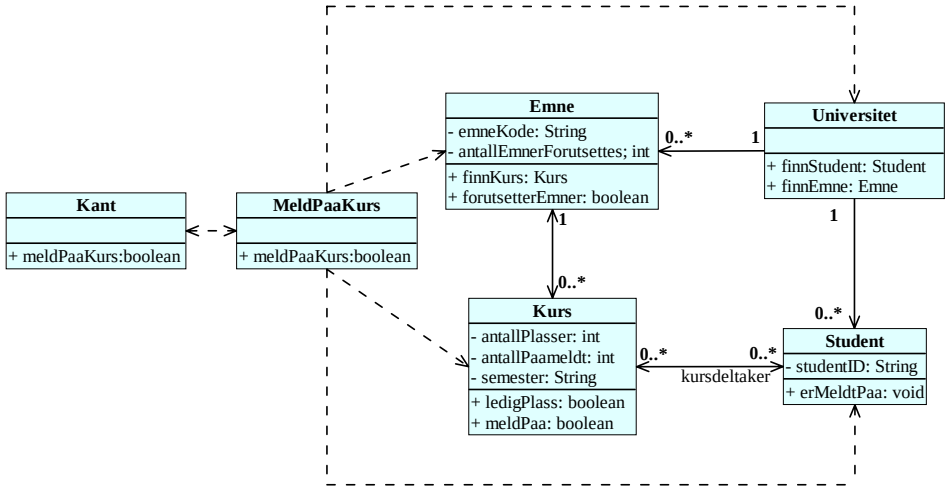
Student <<entity>>	Kurs
Vet min studentID Vet hvilke kurs jeg har tatt Vet hvilke kurs jeg er meldt opp til Vet hvilke kurs jeg står på venteliste på	

Kurs <<entity>>	Emne Student
Vet semesteret som (et kurs i) et bestemt emne holdes Vet antall plasser Vet hvilke studenter som er påmeldt Vet hvilke studenter som står på venteliste	

Normal hendelsesflyt for ”Meld på kurs”

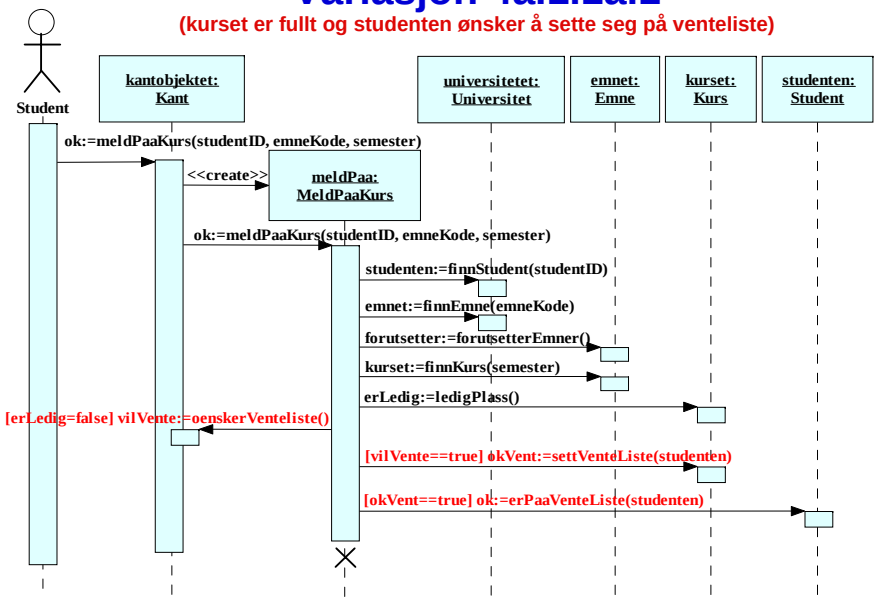


Klassediagram for normal hendelsesflyt

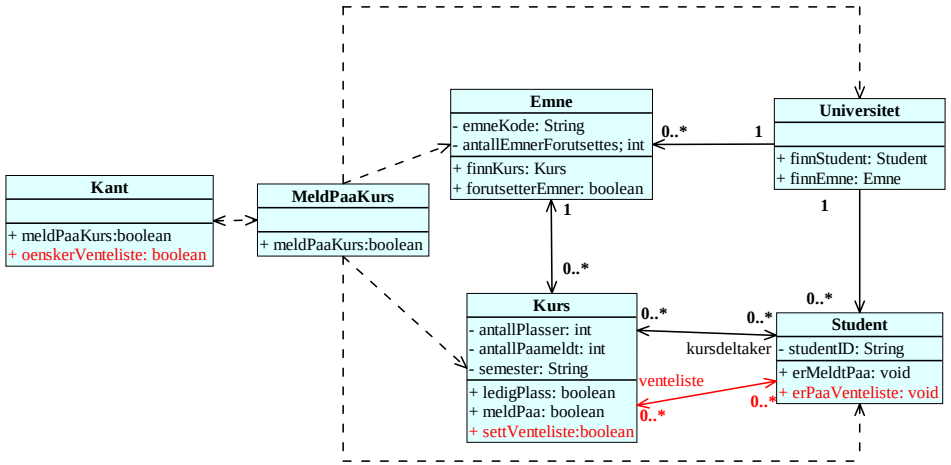


Variasjon 4a.1.1a.1

(kurset er fullt og studenten ønsker å sette seg på venteliste)



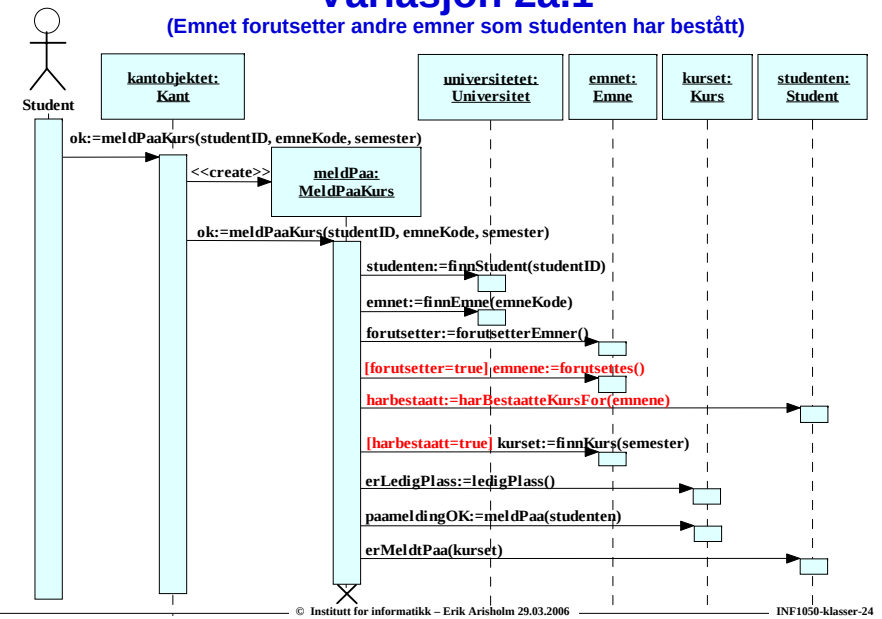
Klassediagram for normal hendelsesflyt og variasjon 4a.1.1a.1



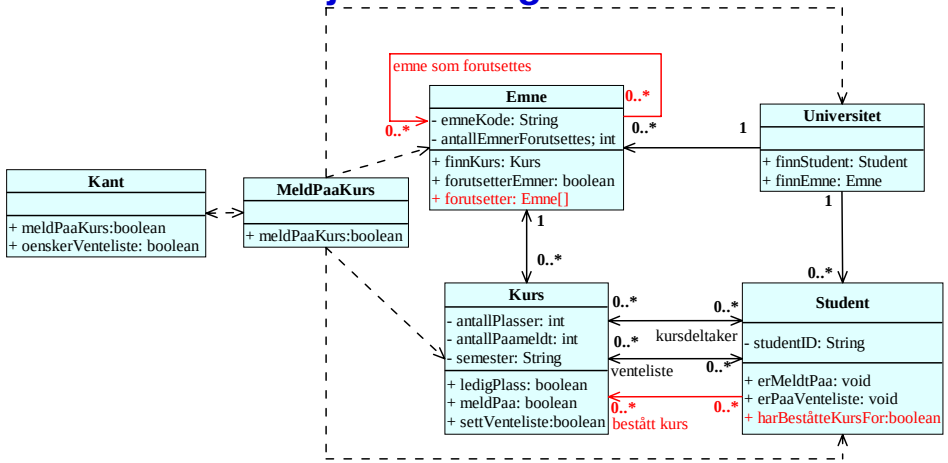
Legger til nye metoder og assosiasjoner for variasjonen 4a.1.1a.1

Variasjon 2a.1

(Emnet forutsetter andre emner som studenten har bestått)



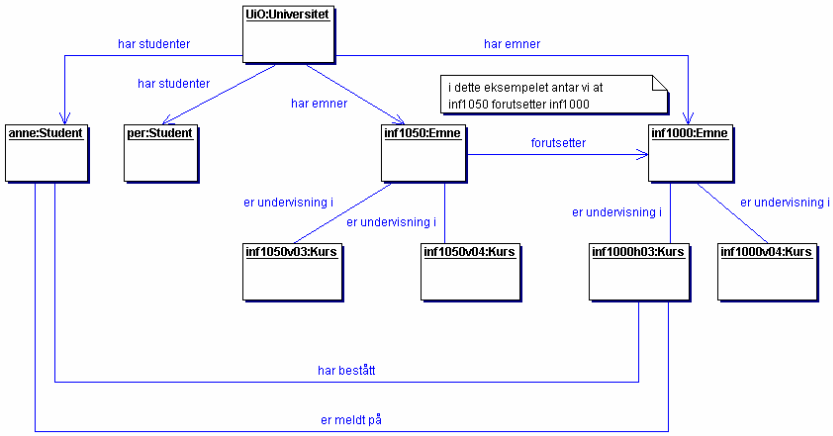
Klassediagram for normal hendelsesflyt, variasjon 2a.1 og 4a.1.1a.1



Legger til nye metoder og assosiasjoner for variasjonen 2a.1

Eksempel objektdiagram for forretningsobjektene

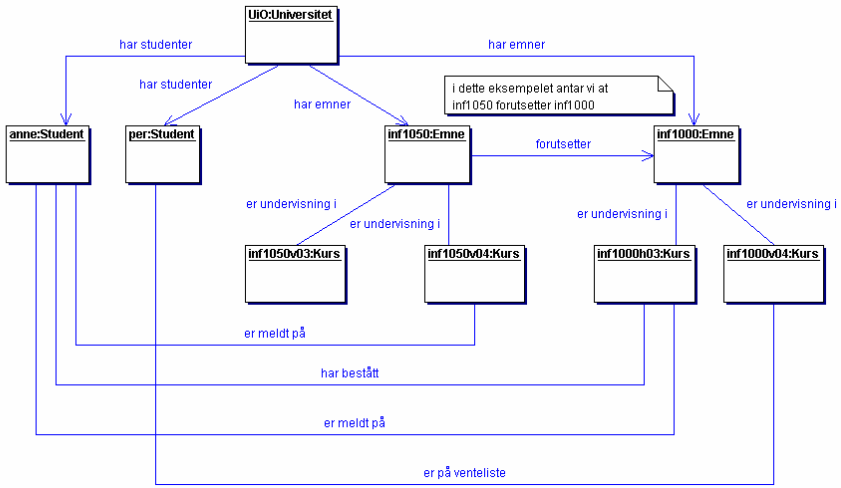
***Før* "Meld På kurs" for Anne og Per:**
Anne (som allerede har meldt seg på og bestått inf1000) ønsker å melde seg på Inf1050 våren04
Per ønsker å melde seg på Inf1000 våren04 (men kurset er allerede fullt)



(For oversiktens skyld vises ikke attributtverdiene til objektene i dette diagrammet)

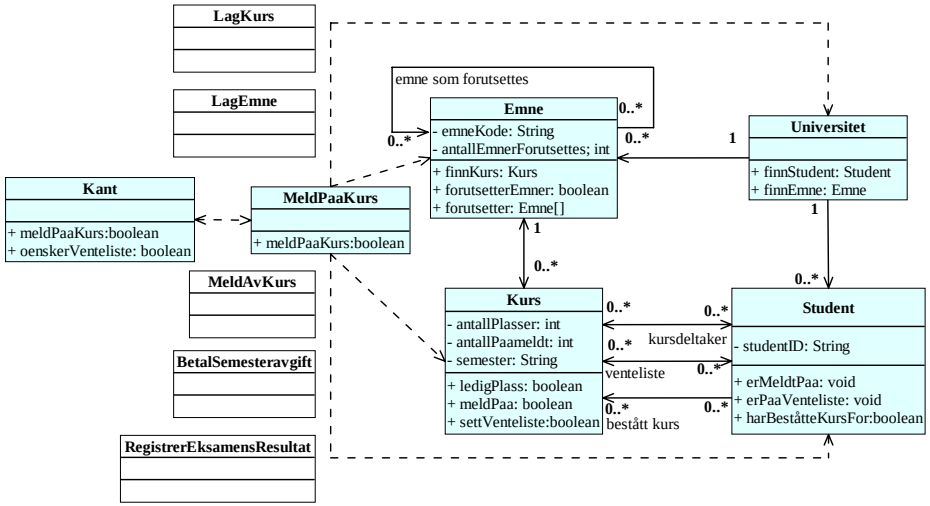
Eksempel objektdiagram for forretningsobjektene

***Etter* "Meld På kurs" for Anne og Per:**
Anne er nå meldt på Inf1050 våren -04,
og Per er satt på venteliste for Inf1000 våren -04.



(For oversiktens skyld vises ikke attributtverdiene til objektene i dette diagrammet)

Klassediagram på systemnivå



De andre bruksmønstrene vil introdusere nye kontrollobjekter og evt. nye forretningsobjekter, samt nye assosiasjoner, metoder og attributter på eksisterende forretningsobjekter. Kantobjektet vil få flere metoder (antar ett kantobjekt i systemet)

Litt om lagring av objektene (persistens)

- Hvilke objekter skal lagres?
  - Forretningsobjektene (f.eks. objektene på forrige objektdiagram)
- Hva lagres?
  - Tilstanden (identitet, attributter og assosiasjoner) til objektene
- Media for lagring
  - Filer: En fil pr objekt? Blir fort ganske uhåndterlig…
  - Objektorientert database:
    - Eksempel: ObjectStore, Poet
    - Tilstanden til forretningsobjektene lagres i en database uten at programmet trenger å tenke så mye mer på det…
  - Relasjonsdatabase: bruke standard klassebiblioteker som skjuler ”oversettingen” (eks: J2EE - Data Access Object)
    - Hente/lagre attributter som tilhører hvert objekt vha SQL
    - assosiasjoner realiseres vha fremmednøkler og evt oppslagstabeller