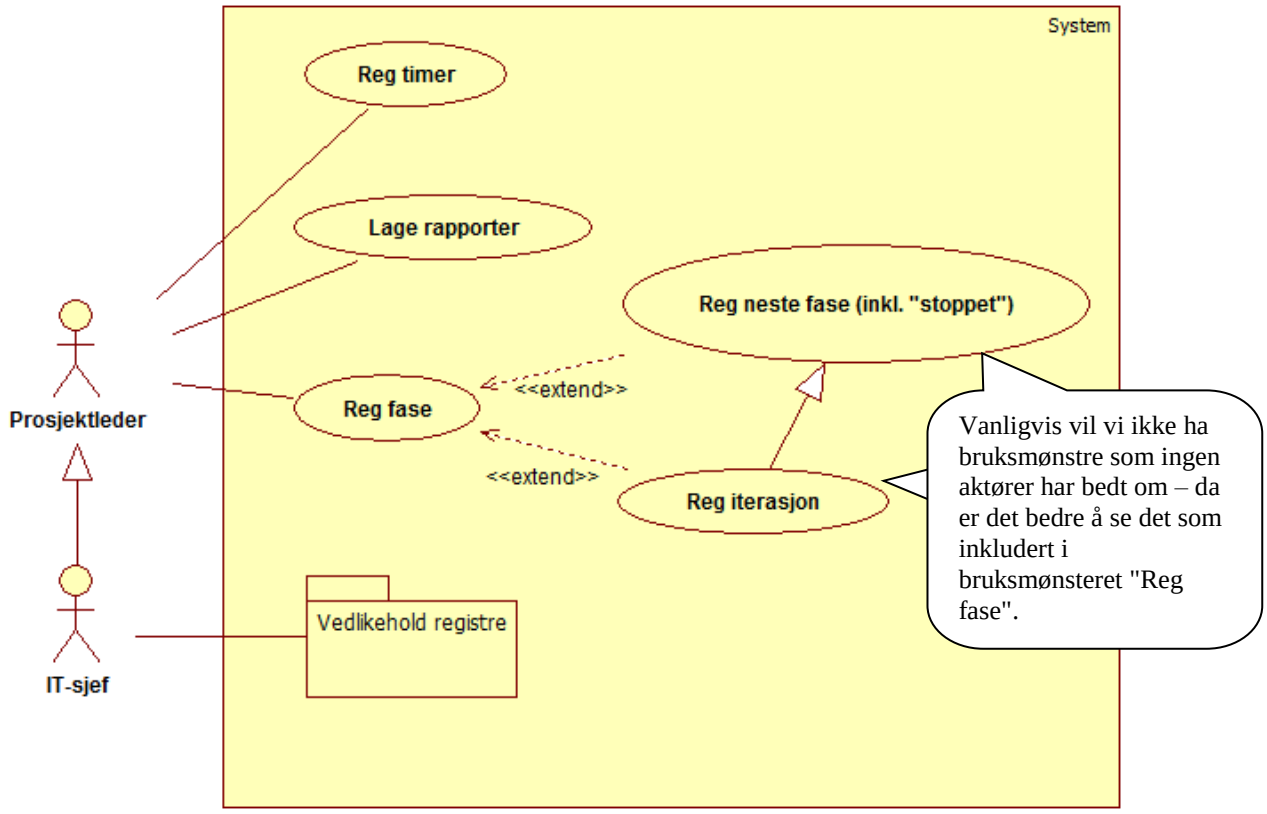
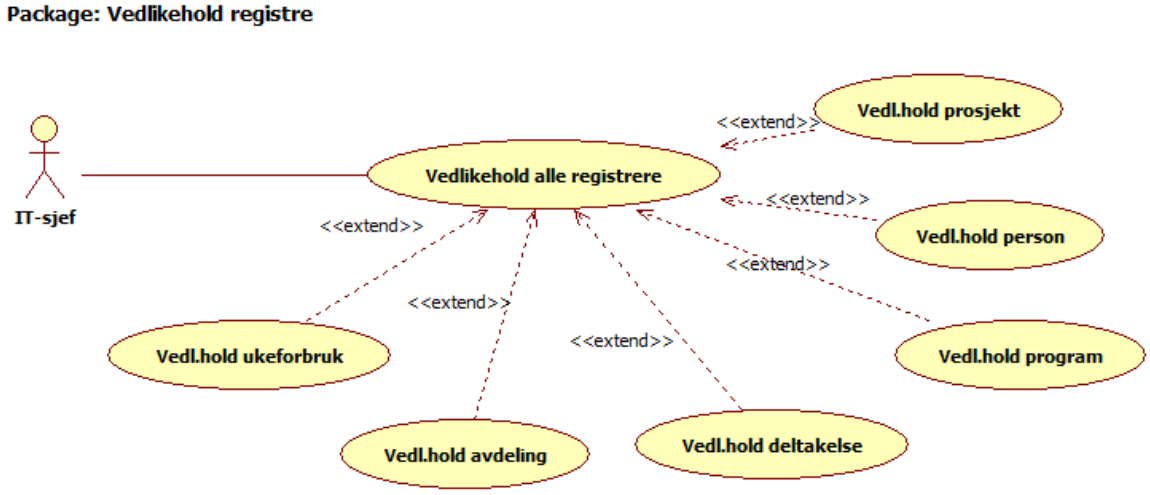
Case INF251 OOAD – Forslag (gammel kurskode) – skrevet av Knut Hansson

Oppgave A

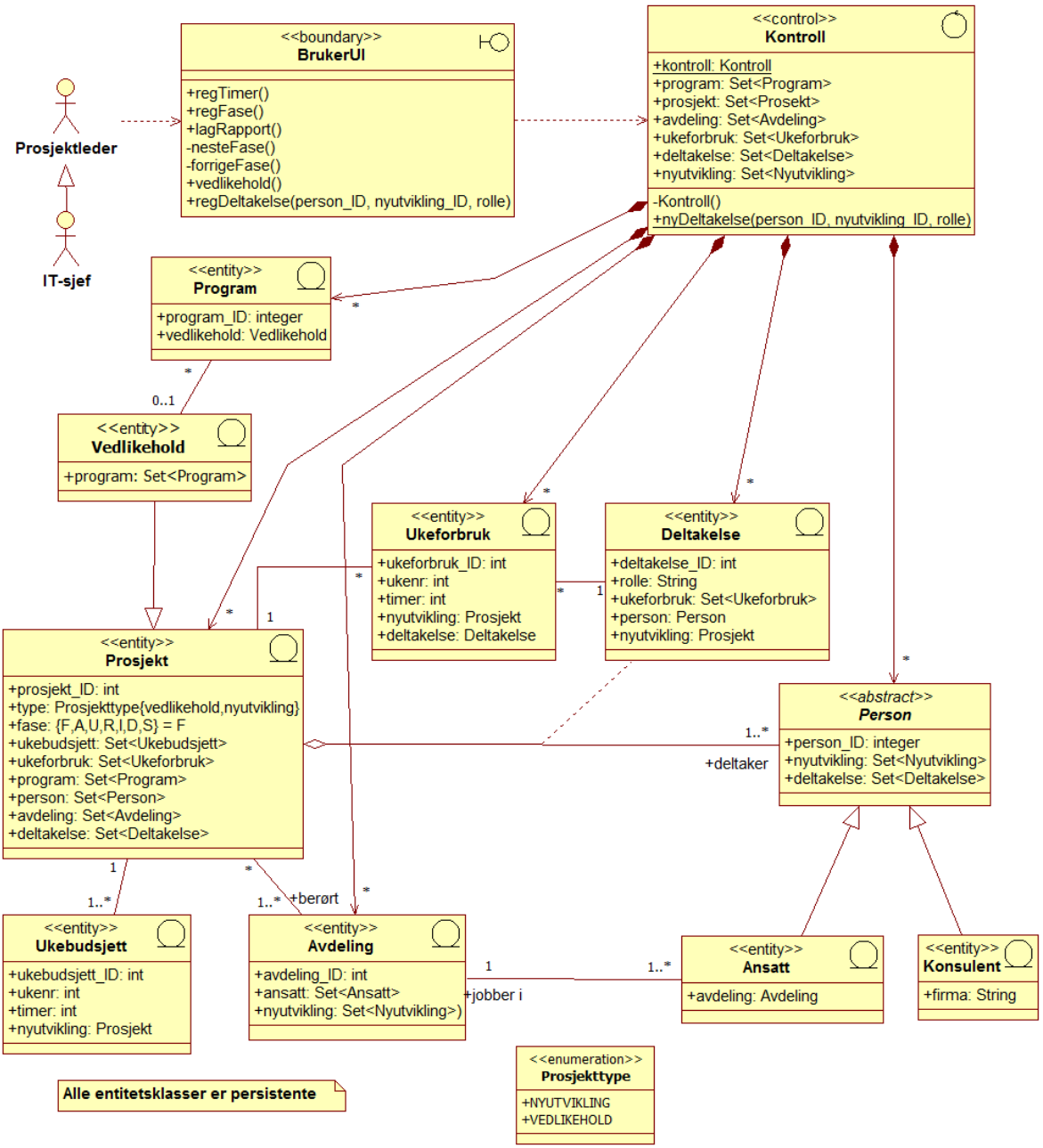


Det er frivillig å ta med systemgrensen – den er uansett underforstått. Vedlikehold av registrene er laget som en pakke (abstrahert) og detaljert på et annet diagram.



De to diagrammene ovenfor er på samme nivå – det øverste skjuler bare noen detaljer i en pakke, mens det nederste skjuler alt annet enn pakkens innhold.

Oppgave B



Jeg tror oversikten her – men kanskje ikke programmet – ville vunnet på å dele "Kontroll" i to: En kontrollklasse for kontrollen og én boundary-klasse som håndterer databasen. I praksis ville man nok (i Java) uansett benyttet JDBC-klasser for databasen.

Jeg mener at det klart er angitt to typer av prosjekter, men de har mye felles, derfor arv. Videre er det angitt to typer deltakere tilsvarende.

Også *aktører* er klasser og her er de tatt med som grenseklasser. En aktør må representeres på "innsiden" av systemet, fordi han/hun tar initiativer som gir en "kaskade" av meldinger i systemet. Det er ikke opplagt at de skal være med i diagrammet (ofte er de det ikke) og da tenker man seg at grenseklassen *BrukerUI* representerer aktørene på innsiden av systemet. Her antyder jeg at disse grenseklassene tar vare på opplysninger om den påloggede brukeren.

De er tegnet med *conic* display (under format) slik de også blir slik når man drar dem inn i klassediagrammet fra use case modellen. Det er en fordel å gjøre det slik, fordi man unngår redundans som man får ved å opprette dem som *ny* klasse i klassemodellen. Hvis man da i bruksmønsterdiagrammene tillegger aktørene egenskaper, vil de automatisk reflekteres i klassediagrammet (og må evt. skjules der).

For å holde orden på domenet *Prosjekttype* har jeg tegnet inn en *enumeration*. Der er "attributtene" de lovlige verdiene som StarUML kaller *literals*. I Java kan vi realisere den slik:

```
public enum Prosjekttype {
    NYUTVIKLING,
    VEDLIKEHOLD,
}
```

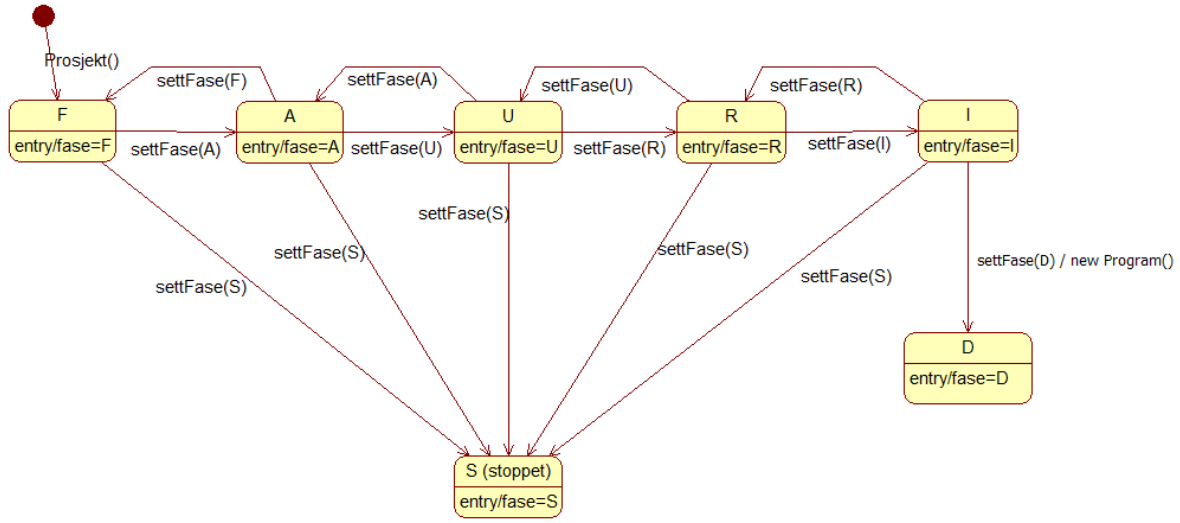
Som vanlig i Java har jeg skrevet navnene med store bokstaver, siden de er konstanter.

Du henviser til en verdi med punktnotasjon:

```
Prosjekttype type=Prosjekttype.NYUTVIKLING;
```

Enumerations kan ikke ha attributter eller metoder, kun konstanter og de kan ikke endres under kjøring.

Oppgave C



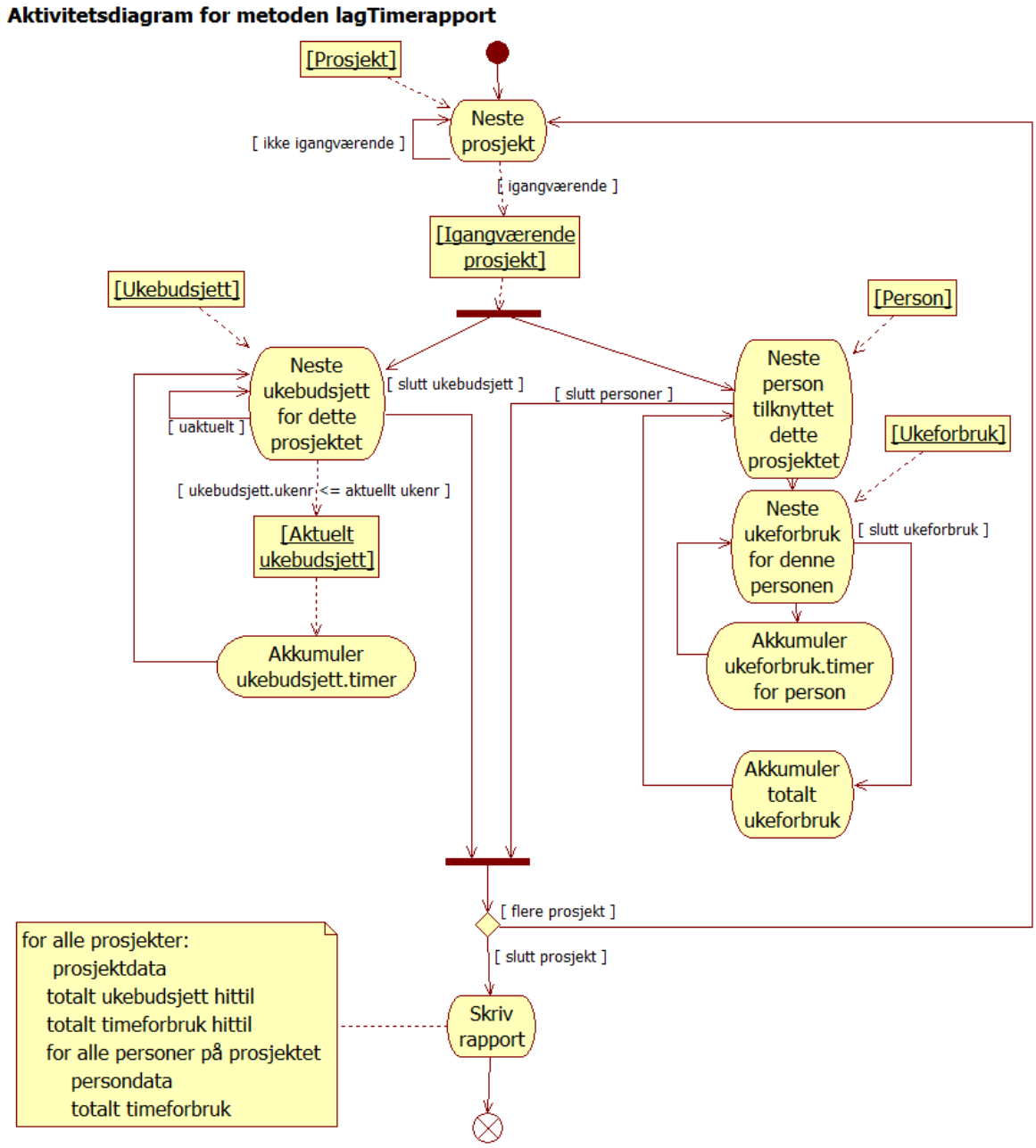
Tilstandsdiagrammer knyttes til ett objekt. Her er det viktig at overgangene trigges av operasjoner som objektet faktisk har.

Det er oppgitt at prosjektene aldri slettes, derfor er det ingen sluttsymbol her. Sluttsymbol vil angi at objektet ikke lenger eksisterer.

Både S og D er terminale og persistente tilstander. Alle de andre er transiente.

Det er angitt i klassediagrammet mitt at initialtilstanden er F.

Oppgave D

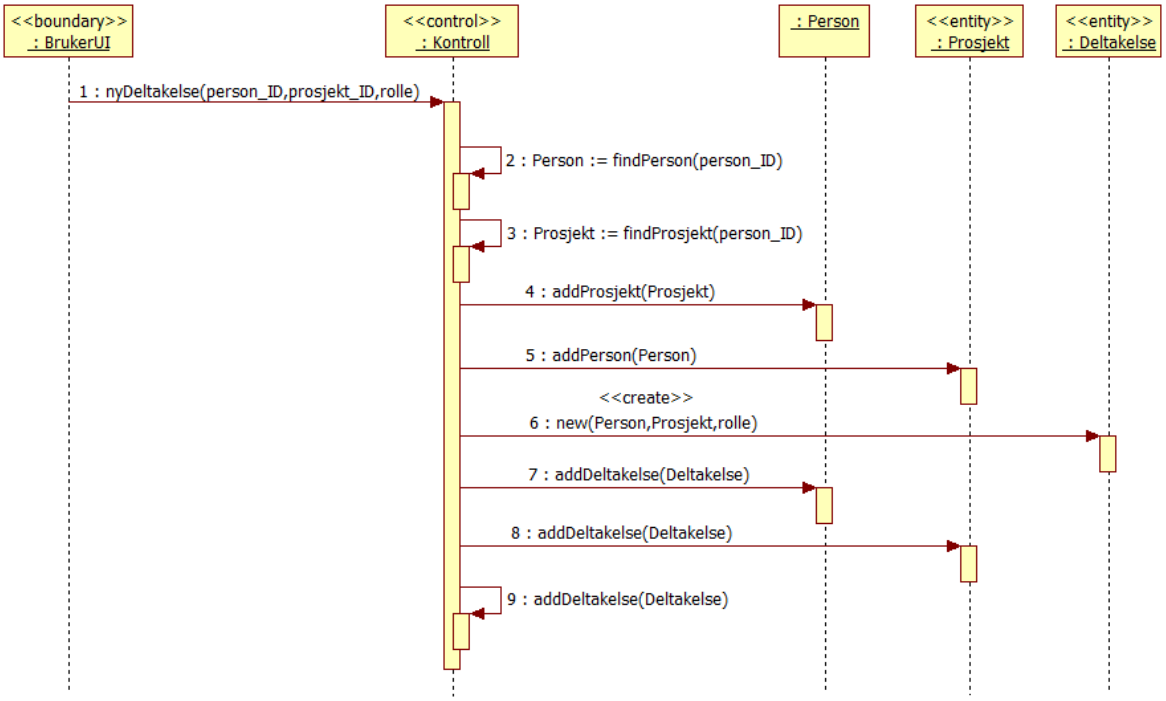


Det sentrale er her å fokuserer på, og dokumentere, *algoritmen* som programmerer skal følge når han/hun programmerer. Derfor skal vi ikke blande inne hva *brukerne* gjør. Jeg mener det helt opplagt blir bruk for både iterasjoner og seleksjoner her.

Jeg legger opp til to tråder, da ukebudsjetter og ukeforbruk kan beregnes uavhengig av hverandre (trådene bruker forskjellige objektsamlinger). Legg merke til hvordan objektsamlingene "strømmer" inn i systemet. De ankommer da ett for ett med en *ForEach* e.l. og behandles også ett for ett. Noen av dem kan velges ut og "strømmer" da videre – andre kommer ikke med.

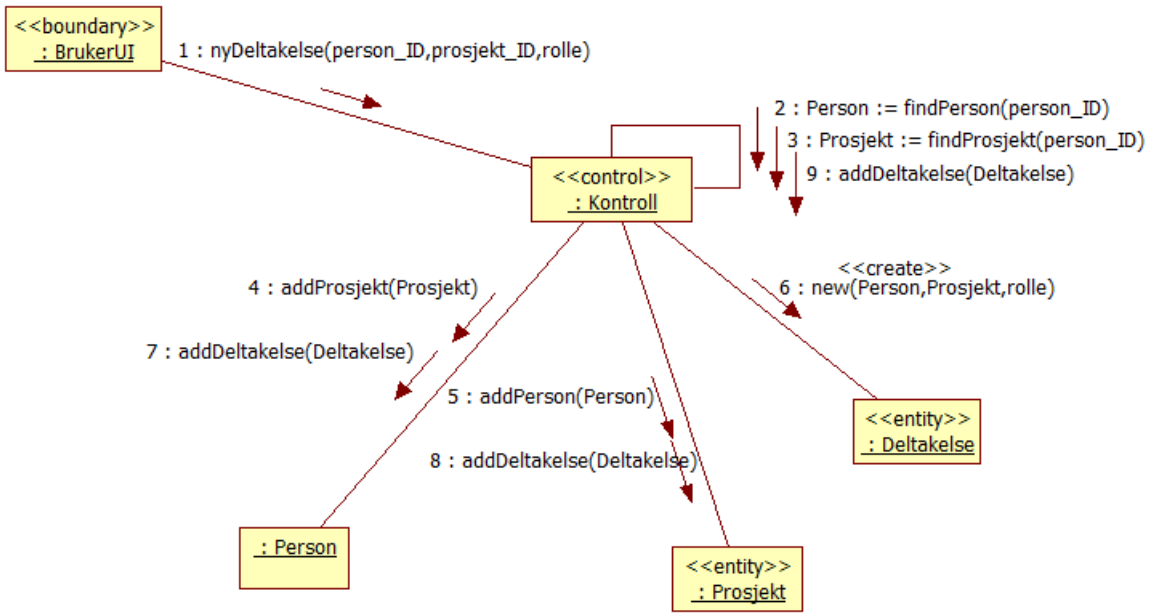
Legg også merke til at en aktivitet kan ha flere "fortsettelser". Da er det angitt når de forskjellige fortsettelsene gjelder med en "guard action".

Oppgave E

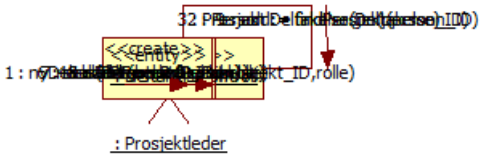


Her har jeg ikke delegert alt nedover. F.eks. kunne man tenke seg at når en person får meldingen *addProsjekt(Prosjekt)* så sender det selv meldingen *addPerson(this)* til *Nyutikling*. Dette har jeg selv forsøkt å programmere noen ganger, og jeg angrer alltid. Man må jo også legge opp til at et annet objekt enn en person sender meldingen, og da skal nyutviklingen legge seg selv til i person med meldingen *addProsjekt(this)*. Dette blir fort indirekte rekursjon. Det går an å unngå det, men det krever mye ekstra kode og kan fort gå feil. Det er mye enklere når et helt annet objekt – her kontrollobjektet – "dirigerer" dette.

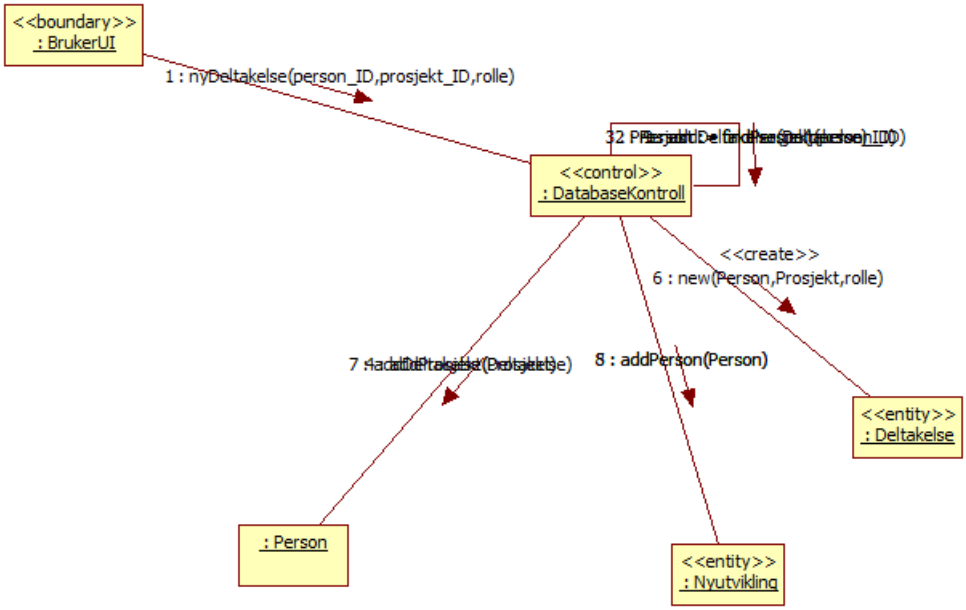
Merk at alle meldingene må være blant dem som objektene tilbyr i henhold til klassediagrammet (eksplisitt eller iflg. standarden). Når kontrollen får meldingen om ny deltakelse, får den bare identifikatoren til person og prosjekt. Derfor må den kalle seg selv for å finne de riktige objektene, og – til slutt – for å legge den nye deltakelsen inn i samlingen sin.



Dette samarbeidsdiagrammet er automatisk generert av StarUML. Jeg får feilmelding, men når programmet lukkes og åpnes igjen, så er det allikevel laget. Det må ryddes betydelig opp i det, da tekster og pile og alle objektene havner oppå hverandre. Slik ser den ut da:



Når objektene er flyttet fra hverandre, ligger fortsatt piler og tekster oppå hverandre:



Fordelen med automatikken er allikevel at de blir semantisk like.