Kompendium for Fellesprosjektet for fagene

TTM4100 - Kommunikasjon
TDT4140 - Systemutvikling
TDT4145 - Datamodellering og Databasesystemer
TDT4180 - Menneske-maskin-interaksjon

Våren 2012

Innhold

1 Introduksjon til fellesprosjektet

1.1 Hva er "fellesprosjektet"?

Velkommen til fellesprosjektet for de fire fagene: TTM4100 - Kommunikasjon, TDT4140 - Systemutvikling, TDT4145 - Datamodellering og databasesystemer og TDT4180 - Menneske-maskin-interaksjon (MMI). Fellesprosjektet utgjør en del av det obligatoriske øvingsopplegget i disse fire fagene, og består av to store innleveringer i løpet av semesteret. Merk at studenter som kun tar noen av disse fire fagene, enten har en begrenset variant av fellesprosjektet, eller har egne obligatoriske øvinger innenfor hvert fag. Det er satt av i alt fire uker til fellesprosjektet – uke 10 og 11 til planlegging og design, hele uke 12 og 13 til realisering og testing. I disse to periodene vil det ikke være forelesninger i de fagene som er involvert i fellesprosjektet.

Fellesprosjektet er viktig av flere grunner: Det gir muligheten til å lære og å praktisere teorien i fire fag, dere får erfaring med prosjektformen for organisering av arbeid, og dere får jobbe i grupper, med alt det måtte føre med seg av gleder og sorger. De fleste vil i ettertid huske fellesprosjektet som et strevsomt og givende fag, det har i hvert fall vært tilbakemeldingen fra tidligere studenter.

Resten av dette kapittelet introduserer oppgaven og gir informasjon om relevante metoder og prosesser. Kapittel 2 forteller hva slags system vi skal utvikle og hva systemet skal brukes til. I kapittel 3 konkretiseres de funksjonelle kravene som stilles til systemet, dvs. alt som brukeren ønsker å kunne utføre. I kapittel 4 beskrives strukturen / arkitekturen til systemet, og andre krav som stilles til konstruksjonen, de ikke-funksjonelle kravene. I kapittel 5 detaljeres kravene som fellesprosjektet setter til sine innleveringer. I kapitel 6 detaljerer vi kraven som fellesprosjektet setter til MMI. Kapitel 7 beskriver viktige momenter for KTN delen av fellesprosjektet, så som funksjonelle krav, krav til innleveringer i dette faget og en beskrivelse av planlagt løsning for kommunikasjonsdelen. Kapitel 8 beskriver databasedelen.

I tillegg har kompendier tre vedlegg som behandler hhv. Use case, use case basert estimering og en kort introduksjon til papirprototyping og brukbarhetstesting.

1.2 Rammer for prosjektet

Prosjektet tar sikte på å gi dere praktisk og realistisk erfaring i prosjektbasert systemutvikling, men rammen for faget tilsier at realismen blir begrenset[1].

I arbeidslivet ville det innledningsvis i et prosjekt være ganske stor usikkerhet mhp. hva som skal gjøres – for eksempel kundens behov og krav til systemet. Mye av utviklingsjobben vil være å analysere kundens problem, vurdere hvilken rolle og funksjoner et system bør ha for å bidra til å løse kundens problemer og om det hele tatt kunne løses på en effektiv måte, og så velge passende utviklingsmetoder, programmeringsspråk, verktøy og lignende. I fellesprosjektet er imidlertid frihetsgradene sterkt reduserende fordi:

- Med for stor frihet vil mange grupper kunne ta feil veivalg i starten, noe som gjør at de ikke greier å fullføre på en fornuftig måte. Selv blant profesjonelle programvareutviklere er en god del prosjekter langt fra vellykkede, og dere befinner dere ennå på et tidlig stadium i en lang utdanning.

[1] I 4. klasse tilbyr vi faget Kundestyrt prosjekt, som er ment å være så realistisk som råd er.

- Hvis et femtitalls prosjekter får lov til å utvikle seg i helt ulike retninger, vil dette kreve veiledningskapasitet som vi verken har finanser eller personellressurser til. Det ville også være vanskelig å oppdrive reelle kunder for så mange grupper.

Følgelig er det i dette prosjektet lagt klare begrensninger på hva og hvordan:

- Det finnes ingen ordentlig kunde. Kravene til produktet er spesifisert på forhånd. Det blir derfor ingen kravspesifikasjonsfase. *Hovedvekten ligger på fasene konstruksjon, implementasjon og testing.*
- Tidsfristene for levering av delprodukter er gitt av oss. Dermed påtvinges alle grupper en viss oppdeling av produktet og fast progresjon i arbeidet, noe som gjør det langt lettere for oss å hjelpe de av dere som får problem på et eller annet tidspunkt.

Prosjektet skal utføres i grupper à 4-6 personer. Gruppesammensetningen bestemmes av oss, uten hensyn til hvem dere helst ville ha ønsket å samarbeide med. Det gis ikke karakter, bare bestått/ikke-bestått (tilsvarende bokstavkarakterene A-E/F), og normalt evalueres hele gruppen under ett. I ekstreme tilfeller av ikke-deltagelse vil det bli aktuelt å stryke enkeltpersoner. Man må bestå fellesprosjektet for å få adgang til eksamen i de involverte fagene. Grupper eller personer som står i fare for å stryke, vil få advarsel to uker før siste innlevering – det skal ikke komme som et sjokk like før eksamen. Viss man gjør sitt beste, behøver man ikke være redd for at man ikke skal greie å bestå. Det lønner seg imidlertid å sikte mot bedre enn akkurat bestått karakter, siden den praktiske kunnskapen vil være nyttig i hvert fag sin eksamen, og i fag og oppgaver senere i studiet.

1.3 Leveranser

Gjennom semesteret skal dere dokumentere en rekke delresultater i form av *innleveringer*. Innleveringene er vist i tabellen nedenfor.

Faser	Innlevering	Beskrivelse	Dato (uke)
Fase 1	FE1	Prosjektplan i henhold til krav i kapitel 4.1	12-03-09 (10)
	FE2	Systemtestplan i henhold til krav i kapitel 4.1	12-03-09 (10)
	FE3	Overordna design	12-03-16 (11)
Fase 2	-	Realisering av systemet	
Fase 3	FE4	Sluttrapport i henhold til krav i kapitel 4.3	12-03-30 (13)

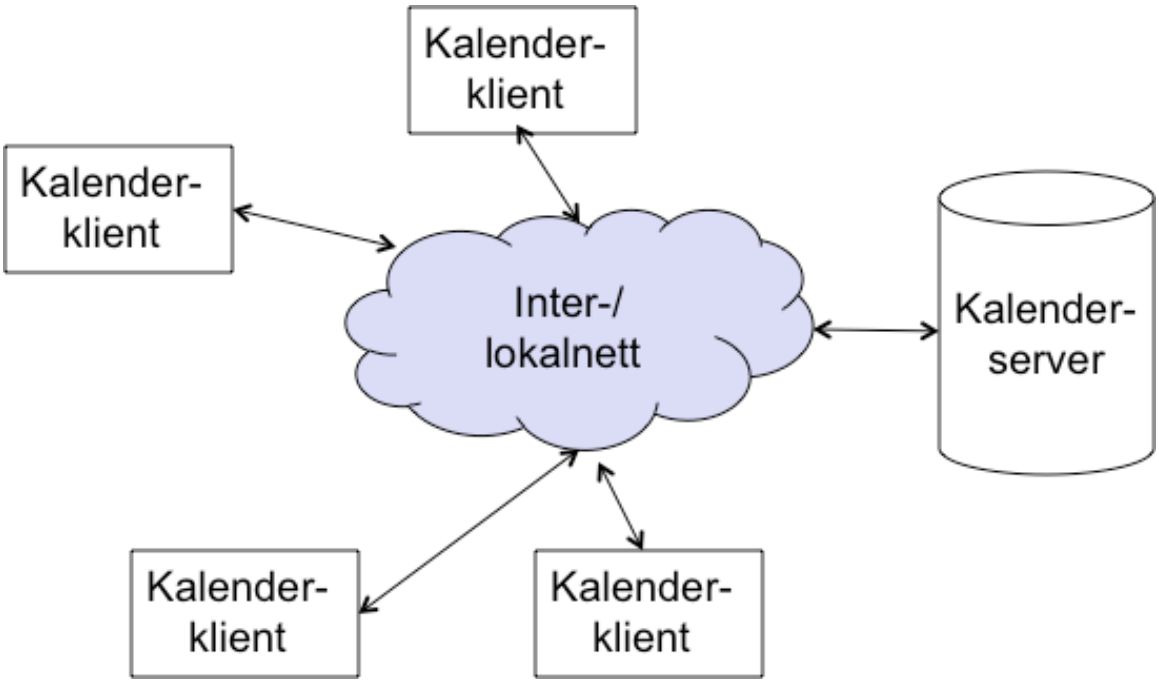
Den informasjonen dere finner i dette kompendiet var riktig i det kompendiet gikk i trykken. Det hender imidlertid at ting forandrer seg, og derfor er det viktig at dere jevnlig sjekker hjemmesidene til fellesprosjektet og de fagene som deltar i dette prosjektet.

Fellesprosjektet:	http://www.idi.ntnu.no/emner/fellesprosjekt
TTM4100 - Kommunikasjon	http://www.item.ntnu.no/fag/ttm4100/
TDT4140 - Systemutvikling	http://www.idi.ntnu.no/emner/tdt4140/
TDT4145 – Datamod og databasesyst	http://www.idi.ntnu.no/emner/tdt4145/
TDT4180 - MMI og grafikk	http://www.idi.ntnu.no/emner/tdt4180/

2 System for elektroniske kalendre

2.1 Oversikt over systemet

Hver ansatt i Firma X skal ha en personlig kalender. Alle kalendrene skal være lagret på en kalendertjener. De ansatte skal ha tilgang til den personlige kalenderen sin med en kalenderklient. Kalenderklienten kjører på en lokalmaskin som kommuniserer med kalendertjeneren over lokalnettet. Mange kalenderklienter kan være koblet opp mot kalendertjeneren samtidig. Sammen utgjør kalendertjeneren og kalenderklientene systemet som skal implementeres.
Figuren under viser en høynivå systemarkitektur over kalendersystemet.



I tillegg til at de ansatte skal kunne planlegge dagene sine med å legge inn avtaler i kalenderen, er hensikten med kalendersystemet å forenkle innkalling til møter. I dag bruker de ansatte i Firma X mye tid på å koordinere intern-møter i organisasjonen. Tiden går med på å finne tidspunkt som passer for alle møtedeltakere, samt å reservere møterom. Kalendersystemet skal administrere innkalling til møter og reservasjon av møterom.

2.2 Krav til kalendersystemet

1. *Logge på*. Ansatte får tilgang til kalendersystemet ved å logge seg på kalenderklienten med brukernavn og passord.

2. *Legge inn avtale*. Ansatte skal kunne legge inn avtaler i den personlige kalenderen sin. En avtale legges inn på avtaledato med et start- og sluttidspunkt, samt en kort beskrivelse av avtalen ("Bil på verksted") og eventuelt sted for avtalen ("Strandveien Auto").

3. *Slette avtale*. Ansatte skal kunne slette en avtale som ligger i den personlige kalenderen sin.

4. *Endre avtale*. Ansatte skal kunne endre på en avtale som ligger i den personlige kalenderen sin. Alle feltene kan endres.

5. *Kalle inn til møte*. En ansatt skal kunne kalle andre ansatte i Firma X inn til et møte. Den som kaller inn til møte kalles en møteleder. En ansatt skal kunne kalle inn til møte på samme måte som han/hun legger ny avtale inn i personlig kalender. I tillegg til feltene for en vanlig avtale, inneholder også møteinnkallingen en liste over innkalte møtedeltakere.

6. *Motta møteinnkalling*. Når en ansatt mottar innkalling til et møte, kan han/hun svare 'Godta' eller 'Forkast'. Ved å svare 'Godta', legges møteinnkallingen inn som en avtale i den innkalte ansattes personlige kalender. Om den ansatte svarer 'Avslag', sendes svar tilbake til møtelederen om at innkallingen ikke er godtatt. Møteleder kan da velge å finne et nytt tidspunkt, avlyse møtet (se under) eller fjerne deltakeren fra innkallingslista.

7. *Endre møteinnkalling*. Møteleder kan endre tidspunkt på en møteinnkalling. Det sendes da beskjed ut til alle møtedeltakerne, som kan svare 'Godta' eller 'Forkast'. Ved å svare 'Godta', endres avtalen i den innkalte ansattes personlige kalender. Ved å svare 'Forkast' sendes beskjed ut til alle innkalte møtedeltakere. Møteleder kan da velge å finne et nytt tidspunkt eller å avlyse møtet (se under).

8. *Avlyse møte*. Møteleder kan avlyse et møte. Det sendes da beskjed til alle møtedeltakerne, og systemet sletter møtet i deltakernes personlige kalender.

9. *Melde avbud for møte*. En ansatt kan melde avbud på en møteinnkalling ved å slette avtalen i sin personlige kalender. Når en ansatt melder avbud, sendes melding til alle de andre møtedeltakerne. Møteleder kan da velge om møtet skal avlyses eller om han/hun skal endre tidspunkt på møtet.

10. *Reservere møterom*. I stedet for å skrive inn sted for en avtale eller et møte, skal brukeren kunne reservere møterom. Kalendertjeneren skal lage en liste med tilgjengelige møterom (mao. ikke reserverte) i tidsperioden for avtalen/møtet. Brukeren kan da velge møterom fra denne listen. Om en avtale med booket møterom slettes, skal reservasjonen slettes på kalendertjeneren. Det samme gjelder for møter som avlyses.

11. *Visning*. Kalenderklienten skal vise en ukekalender der alle avtaler og møter i den ansattes personlige kalender vises. Det skal være enkelt å bla mellom ukene.

12. *Spore møteinnkallinger*. Kalenderklienten skal indikere i ukekalenderen om a) en møteinnkalling venter på svar fra en eller flere deltakere, b) en eller flere møtedeltakere har avslått møteinnkalling, eller c) om alle innkalte har godtatt møteinnkallingen.

13. *Vis flere kalendre*. Det skal være mulig å vise andre ansattes avtaler sammen med sine egne i kalenderklienten.

2.3 Scenarier

Scenariene er ikke en del av kravspesifikasjonen, men skisserer noen typiske bruksscenarier for kalendersystemet.

1. En ansatt har invitert tre forretningsforbindelser til samtaler i Firma X sine lokaler. Den ansatte legger inn dette som en avtale i kalenderen sin, og får kalendersystemet til å booke et ledig møterom som er passe stort for fire deltakere.

2. Jens kaller Beate, Morten og Finn til møte. Kalendersystemet førstkommende fredag klokka 12 til 14. Kalendertjeneren sender melding til de tre Jens har kalt inn til møte. Morten er logget på systemet med en klient, og mottar øyeblikkelig melding om møteinnkallingen. Beate og Finn mottar meldingen neste gang de logger på.

   Beate avslår møteinnkallingen fordi hun har en annen avtale på det tidspunktet. Morten og Finn godtar innkallingen. Først forsøker Jens seg med å endre tidspunktet på møtet. Kalendertjeneren sender ny melding ut til møtedeltakerne. Denne gangen godtar både Beate og Finn møteinnkallingen, men Morten avslår. Jens velger å slette Morten fra møteinnkallingen.

Etter at Beate og Finn har godtatt møteinnkallingen blir Jens syk. Han sletter møtet fra kalenderen sin. Kalendertjeneren sender da melding til de to andre deltakerne om at møtet er avlyst. Beate og Finn mottar begge denne meldingen neste gang de logger seg på kalendersystemet.

3 Arkitektur

Dette kapitlet beskriver krav til konstruksjon eller systemdesign for fellesprosjektet. Konstruksjon av et system skal i prinsippet ikke påvirke de funksjonene som systemet har, men er relatert til de ikke-funksjonelle krav, for eksempel utvidbarhet, vedlikeholdbarhet, ytelse og skalerbarhet. Flerbrukerstøtte er et overordnet ikke-funksjonelt krav, dvs. at mange brukere skal kunne få tilgang til og endre de samme underliggende dataene, fra ulike maskiner på et nettverk.

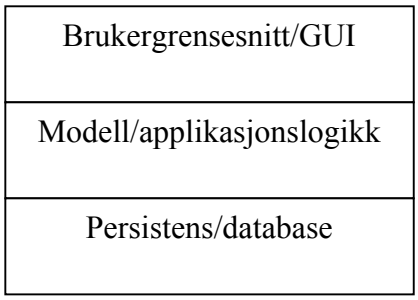
Ved siden av de spesifiserte ikke-funksjonelle kravene, er det også en del generelle egenskaper ved konstruksjonen som er viktig, for eksempel at funksjoner har definerte grensesnitt i forhold til hverandre og at funksjoner en ønsker å endre på uavhengig av hverandre, ikke er for tett viklet sammen. Det er spesielt to typer koblinger vi er opptatt av å løse opp, nemlig koblinger mellom:

1. Applikasjonsdata (modell) og grafisk brukergrensesnitt (GUI)
2. Applikasjonsdata, slik de representeres mens applikasjonen er i bruk, og deres persistente representasjon på fil eller i en database.

Den andre av disse er konstruksjonsmessig knyttet til kravet om flerbrukerstøtte, som vi etter hvert skal se i de etterfølgende kapitlene. Kapittel 3.1 beskriver en overordnet systemarkitektur for de som har både SU, DB og KTN. For de som ikke tar DB, KTN og / eller MMI, vil det være noen endringer som er beskrevet i **Kapittel 3.5.**

3.1 Trelags applikasjonsarkitektur

Det er vanlig å dele applikasjoner i tre deler, med hver sin funksjon (se Figur 1):



Figur 1 Trelagsapplikasjon

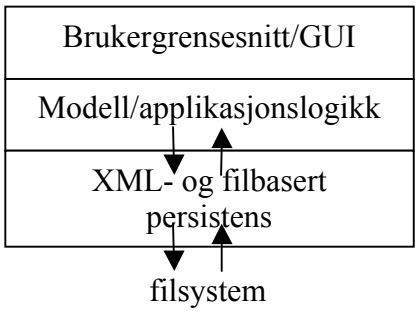
1. *Brukergrensesnittet*, i vårt tilfelle et *GUI*, er den delen som brukeren interagerer direkte med gjennom bl.a. skjerm, mus og tastatur. GUI gir brukeren mulighet for å navigere i og se på data og gir tilgang til funksjoner for å endre dataene. Strukturen på GUI er nært knyttet til strukturen til de dataene, slik den er beskrevet i f.eks. et UML klassediagram, men først og fremst basert på hva brukeren ønsker å gjøre. Implementasjonen av GUI gjøres ofte vha. et GUI-rammeverk og som regel i samme programmeringsspråk som applikasjonen forøvrig, i vårt tilfelle Java og Swing.
2. *Modellen* er alle objektene i den kjørende applikasjonen som brukeren er interessert i å få tilgang til. Disse objektene eksisterer uavhengig av om GUI viser dem, og det er ikke umulig eller unaturlig å ha flere typer brukergrensesnitt som gir tilgang til de samme applikasjonsdataene, eller kun deler av disse. Modellen inneholder både data og regler for hvordan disse kan manipuleres, og sistnevnte kalles ofte *applikasjonslogikk*. Modellen vil typisk være beskrevet vha. UML klassediagram, som et ledd i analysearbeidet i forbindelse med. kravspesifikasjonen. Dersom en bruker et objektorientert språk som Java, vil både data og logikk implementeres vha. klasser.
3. *Persistens*delen håndterer lagring av modellen mellom hver kjøring av applikasjonen, og er typisk støttet av en *database* eller et filsystem. Igjen er det ikke umulig eller unaturlig å tenke seg at de samme dataene kan lagres på ulike vis av en og samme applikasjon, f.eks. som en XML-fil, i en XML-database eller i en relasjonsdatabase. Førstnevnte passer bra for dokumentorienterte enbrukerapplikasjoner, sistnevnte passer bra for flerbrukerapplikasjoner med mer komplekse data og behov for transaksjonshåndtering. Beskrivelsen av de persistente dataene er avhengig av den underliggende mekanismen, og kan være XML-dtd og –skjema og ER-diagram og SQL-skript.[2]

Ved konstruksjon er det viktig å tenke på både den interne strukturen i hver av de tre delene, som gjerne følger bestemte regler eller mønster for konstruksjon, og grensesnittet og samspillet mellom dem. Den interne strukturen i brukergrensesnittet og forholdet mellom brukergrensesnittet og modellen er en sentral del av MMI-faget. DB-faget tar for seg forholdet mellom applikasjonen og databasebasert persistens og struktur og egenskaper for databasen. Dersom disse tre delene av applikasjonen er skilt fra hverandre med et nettverk, har KTN-faget sitt å si om hvordan dette bør håndteres. SU-faget har den modellen/applikasjonslogikken og den overordnede strukturen eller *arkitekturen*, som sitt anliggende.

[2] Det finnes også såkalte XML-databaser som tilbyr en mellomting.

3.2 Fra enbrukerapplikasjon til klient-tjener-arkitektur

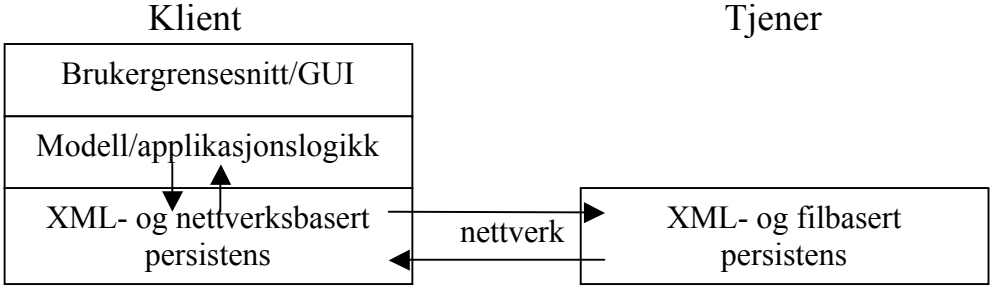
Applikasjonsarkitekturen som er beskrevet over, er et greit utgangspunkt for å konstruere en enbrukerapplikasjon. I dette prosjektet kan en forestille seg at en i første omgang støtter filbasert lagring med XML som format, som illustrert i Figur 2.



Figur 2 Enkel enbrukerapplikasjon

I en dokumentorientert applikasjon vil hele modellen bli lagret som en transaksjon, og grensesnittet mellom modellen, dvs. de to pilene, vil være nokså enkelt. Den venstre pilen representerer funksjonen modell-til-XML-oversetting, mens den høyre pilen tilsvarer den motsatte funksjonen XML-til-modell-oversetting. Merk at funksjonene håndterer kun hele modeller, så det er ikke behov for å oversette delmodeller i den ene eller andre retning. Det kan selvsagt være lurt å bryte funksjonene ned i mindre deler, basert på modellens struktur, men dette er konstruksjonsdetaljer som ikke påvirker grensesnittet representert ved pilene.

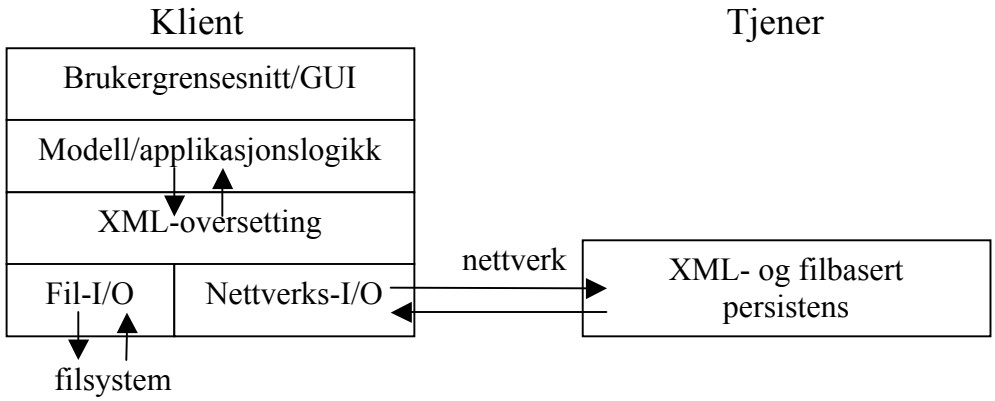
En ulempe med denne applikasjonsarkitekturen er at XML-filer ikke kan leses eller skrives over nettverket (med mindre en utnytter nettverkstøtten i filsystemet), f.eks. http- eller ftp-URL'er. Det skal imidlertid ikke så mye endring til for å håndtere dette, som vist i Figur 3.



Figur 3 Enkel arkitektur med tjenerbasert lagring over nett.

Her vil klientapplikasjonen være så godt som uendret. Den eneste forskjellen er at lesing og skriving av XML-dokumentet skjer over nettverket, vha. av en eller annen protokoll. Dersom en bruker URL-er til å adressere tjeneren og Java sin innebygde støtte for lesing og skriving til URL-oppkoblinger, så er det lite omkoding som trengs. I tillegg er det et stort poeng at endringen er intern i persistensdelen, så GUI og modellen er ikke berørt, kanskje bortsett fra dialogelementer for visning og innskriving av URL-en.

Tjeneren vil i dette tilfellet være nokså enkel, da den kun skal kunne ta i mot en oppkobling og kunne lese fra og skrive til nettverket og henholdsvis skrive til og lese fra fil.

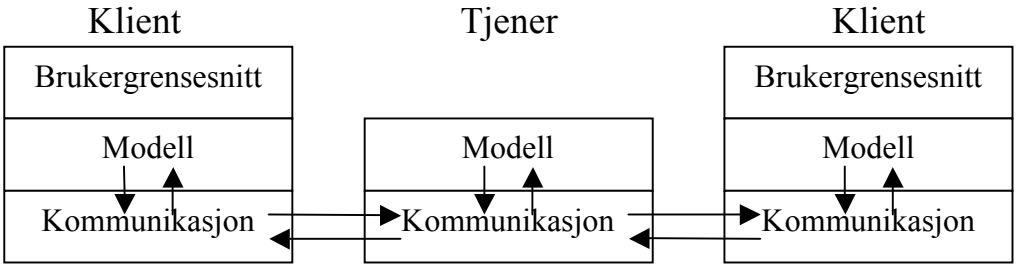


Figur 4 Arkitektur med fil- og nettverksbasert lagring.

Dersom en deler opp persistensdelen i en del som håndterer oversettingen til og fra XML og en del som håndterer lesing og skriving, er det relativt enkelt å få til en kombinasjon av disse to. Dette er vist i Figur 4.

Dette krever at overgangen mellom XML-oversettingen og lesing/skriving av XML-dokumentet er tydelig, f.eks. definert i et eget Java-grensesnitt og at en har klasser for filbasert og nettverksbasert I/O som implementerer dette grensesnittet. Dette tilsvarer forøvrig arkitekturen til først inkrement, med HTTP som nettverksprotokoll.

Selv om vi i de to siste figurene har innført en tjener, er tjeneren såpass enkel at arkitekturen ikke håndterer at flere brukere får tilgang til de samme dataene. Rollen til tjeneren er foreløpig kun knyttet til lagring og ikke til delt tilgang for flere klienter til felles data. For å få det til er det vesentlig at tjeneren også har et modell-lag og mekanismer for å håndtere delt tilgang. Figur 5 skisserer en arkitektur for å håndtere dette, men figuren forteller bare en del av historien.



Figur 5 Arkitektur med delt modell på tjeneren.

Merk at modellene kan godt være implementert av akkurat de samme klassene og vi kan godt bruke XML som overføringsformat. I tillegg til å innføre et modell-lag i tjeneren, er det viktig å definere regler for samspillet mellom tjenerens modell og modellen i hver klient:

1. Tjenerens modell er den offisielle modellen.
2. Klientene må ved hver vesentlige endring lokalt, overføre endringen til tjeneren, for å unngå inkonsistens mellom lokal og offisiell modell.
3. Tjeneren må, når den mottar endringer fra en klient, integrere endringene i sin modell, og videreformidle endringene til de andre klientene.

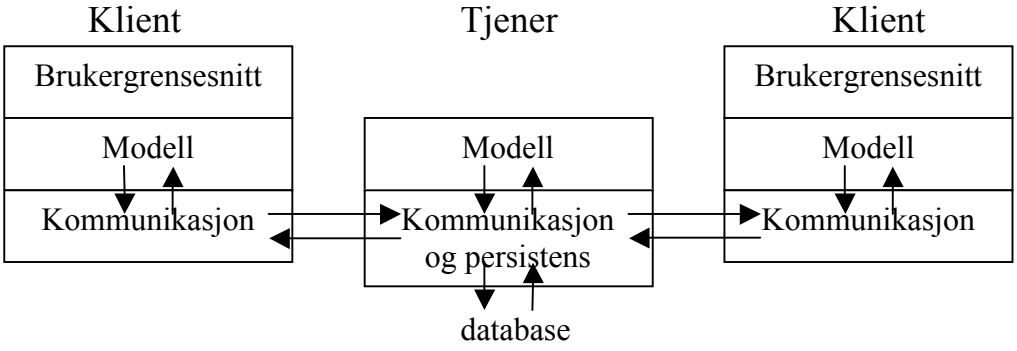
4. Klientene må være beredt til å motta endringer fra tjeneren og integrere disse i sin egen lokale modell.

Kommunikasjonen mellom tjeneren og klientene har altså som formål å sikre konsistens mellom lokale modeller og den tjenerens offisielle, og dataene som sendes over nettverket må være tilpasset dette formålet, for eksempel:

- Både tjeneren og klientene må kunne overføre *fragmenter* av modellen og ikke bare hele modellen som i enbrukerapplikasjonen
- Overføringen må inneholde informasjon om hva slags endring som skal gjøres i den andre enden, f.eks. slette, oppdatere eller opprette objekter
- En må kunne angi hvilke deler av den offisielle modellen som skal endres, f.eks. hvilket attributt som skal endre, hvilket objekt som skal slettes eller hvilken relasjon som skal opprettes

Det som overføres i begge retninger mellom klient og tjener kan sees på som en kommando med tilhørende data, og en viktig del av designarbeidet er å definere kommandoene slik at dekker behovene over og bestemme hvordan de skal kodes i XML og formidles over HTTP.

Som vi ser er persistenslaget omdøpt til kommunikasjon i figuren over, men det betyr ikke at persistens er mindre viktig. Vi må fortsatt sikre at dataene overlever når applikasjonen avslutter, så persistensmekanismen må gjeninnføres, slik vi har vist i Figur 6.



Figur 6 Tjeneren sikrer data vha. databasebasert persistens.

For å sikre at modellen kan gjenopprettes hvis tjeneren krasjer, er det viktig at databasen blir oppdatert i takt med tjenerens modell. Hvilken som oppdateres først, modellen eller databasen, er en smakssak, men begge bør gjøres før klientene oppdateres, for å minimere muligheten for at krasj underveis. Det overordnede poenget er at det er databaseinnholdet som er grunnlaget for å gjenopprette modellen ved krasj eller vedlikehold av tjeneren. Derfor er databasen til syvende og sist mer ”offisiell” enn modellen i den kjørende tjeneren, siden det er den vi sitter igjen med når strømmen slås av.

3.3 Arkitektur og Java-programmering

Overgangen fra abstrakte arkitekturfigurer som vist over, til praktisk Java-koding kan kanskje virke vanskelig. Dette får dere mer informasjon om på forelesningene, men her er to vesentlige tips å ta med seg:

1. Arkitekturen er en funksjonell nedbrytning av applikasjonen, og denne nedbrytningen kan være grei å bruke når en skal definere Java-pakkestrukturen for prosjektet. F.eks. kan en ha en A-pakke på toppen med pakker for GUI, modell og persistens under. Alle GUI-

klasser vil ha A.gui som pakkeprefiks, for eksempel A.gui.ProduktPanel, mens modellklassene vil ha A.modell som pakkeprefiks, for eksempel A.modell.Produkt.

2. Definer koblingen mellom lagene i arkitekturen vha. Java-interface, for eksempel A.persistens.ModellLeser og A.persistens.ModellSkriver for lesing og skriving av modellen. Implementasjonen av disse grensesnittene kan godt ligge i egne underpakker, f.eks. A.persistens.fil.Model.ModellLeserImpl og A.persistens.fil. ModellSkriverImpl.

3.4 Persistens og håndtering av data

Data skal lagres i en relasjonsdatabase, som tjeneren kommuniserer med ved hjelp av JDBC. Alle endringer som skjer i systemet eller et prosjekt, f. eks. at administratoren legger inn en ny oppgave eller at en utvikler registrerer en ny erfaring, skal føre til at databasen oppdateres for å reflektere den nye situasjonen. Tjeneren skal kunne krasje eller skrus av, og deretter skrus på igjen, uten at data går tapt. Tjeneren skal lese inn data fra databasen når den starter opp.

Data som tjeneren tar imot fra klienten er strukturert som XML. Denne skal derfor analyseres av tjeneren, og informasjonen i den legges inn i databasen på en hensiktsmessig måte. En del av prosjektoppgaven er å utforme en hensiktsmessig datamodell, implementere denne i en Oracle-database, og lage et program som fyller den med data på grunnlag av XML-filen.

Tjeneren og databasen skal kommunisere vha. JDBC, som er standardbiblioteket for databasekommunikasjon i Java. JDBC vil bli forelest i DB-faget.

All informasjon som tjeneren sender til klientene/gateway skal være korrekt i henhold til databasen. En mulig måte å gjøre dette på, er å utføre spørringer mot databasen hver gang. Å cache deler av data i tjeneren kan være mer effektivt, men kan også øke kompleksiteten. Dere står imidlertid fritt til å implementere dette slik dere finner det for godt.

3.5 Endringer i krav for grupper som ikke har alle fagene

- For de som ikke har DB skal all persistens bli ivaretatt av filsystemet.
- De som ikke har MMI må selv lage et enkelt brukergrensesnitt for å kunne teste systemet. Dette brukergrensesnittet kan gjerne være kommandobasert.
- Alle krav i kravlista relatert til nettverk utgår for de som ikke har KTN.

3.6 Kjerneapplikasjonen

Kjerneapplikasjonen er en enkel adressebok der brukeren kan legge til, fjerne og redigere en liste med personer og tilhørende personalia som fødselsdag og e-postadresse.

3.6.1 Installasjon

Kjerneapplikasjonen er gjort tilgjengelig som ei zip-fil på Web. Fila kan lastes ned fra http://www.idi.ntnu.no/emner/fellesprosjekt/kjerne.zip. Gjør følgende for å få tilgang til kildekoden fra zip-fila i Eclipse:

1. Pakk ut fila kjerne.zip (dobbeltklikk på fila i Utforsker) i en midlertidig katalog (f.eks. m:\tmp\). Denne midlertidige katalogen kan ***ikke*** være den samme som du velger som Workspace i steg 2. Fila oppretter en ny katalog, kjerne, i den valgte katalogen (f.eks. m:\tmp\kjerne\)

2. Start Eclipse. I *Workspace Launcher* som kommer opp ved oppstart velger du Workspace-katalogen der du har kildekoden din (f.eks. m:\eclipse). Opprett en katalog om du ikke allerede har en katalog til kildekoden din.

3. Opprett et Eclipse-prosjekt for kjerneapplikasjonen

   - I Eclipse, opprett et nytt prosjekt for kjerneapplikasjonen ved å velge File -> New -> Project. Dialogboksen *New Project* kommer opp.

   - I dialogboksen *New Project* velger du *Java Project* og klikker *Next>*.

   - Kall prosjektet ”kjerne” (føres i innskrivingsboksen *Project name*) i neste skjermbilde, og klikk *Finish*.

4. Importer kildekoden til kjerneapplikasjonen inn i Eclipse-prosjektet

   - Importer kjerneapplikasjonen ved å velge File -> Import. Dialogboksen *Import* kommer opp.

   - I dialogboksen velg 'File system' under *Select an import source:*, og klikk *Next>*

   - I innskrivingsboksen *From directory* skriver du enten inn stien der du pakka ut kjerne.zip (f.eks. m:\tmp\kjerne\) eller bruker *Browse*-knappen for å finne katalogen.

   - Klikk på knappen *Select all* og så *Finish*.

   - Svar *Yes to all* på eventuelle spørsmål som dukker opp under importeringa av kildekoden.

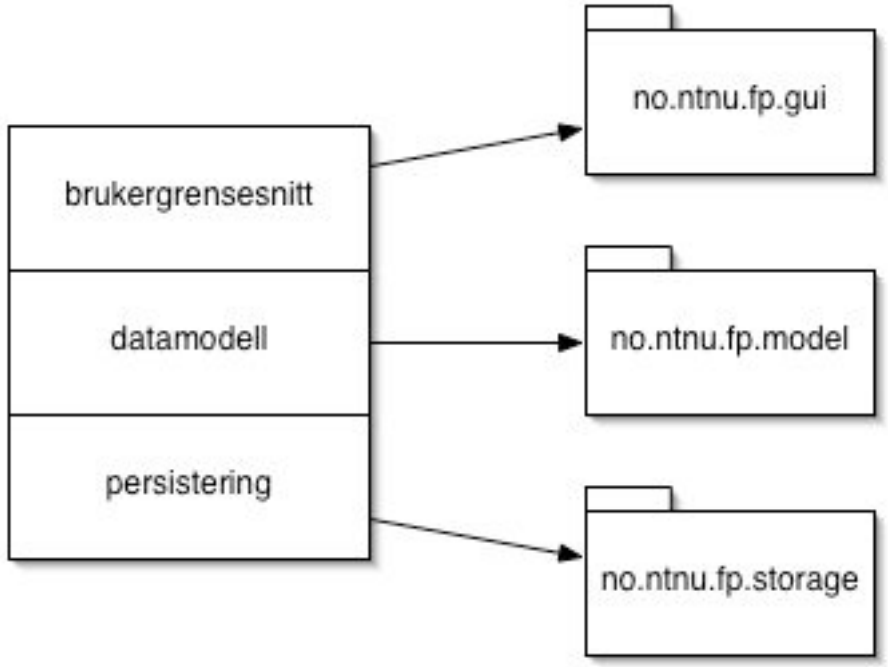
Du har nå tilgang til kildekoden i Eclipse.

3.6.2 Arkitektur

Kjerneapplikasjonen bygger på en trelagsarkitektur. Det vil si at applikasjonen er delt inn i tre lag: et lag for brukergrensesnitt, et lag for datamodell og et lag som tar vare på persisterings.

Figur 7 Lagdeling av applikasjonen

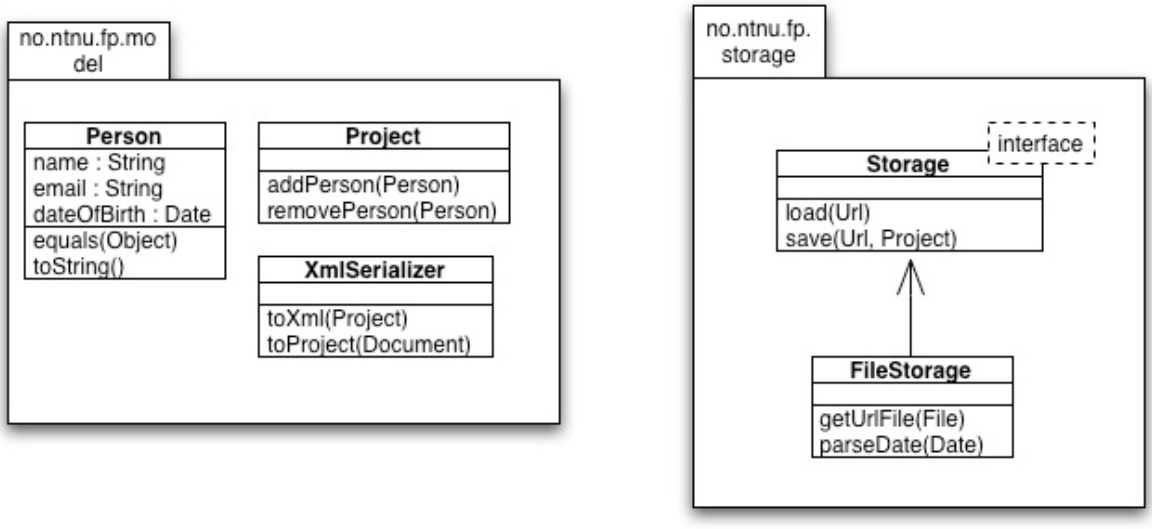
Hvert av lagene i applikasjonen tar seg av ulike funksjoner. Alle Java-klassene som implementerer brukergrensesnittet og funksjonaliteten forbundet med brukergrensesnittet, hører hjemme i brukergrensesnittlaget. Klassene som implementerer datastrukturene i kjerneapplikasjonen og tilhørende funksjonalitet, hører hjemme i datamodell-laget. I persistersistens-laget ligger klassene som tar seg av å lagre og lese inn data fra disk. I praktisk Java har vi delt koden for kjerneapplikasjonen inn i Java-pakker. En Java-pakke for hvert av lagene i applikasjonen:



Figur 8 Lagdeling med korresponderende Java-pakker

Hensikten med trelagsarkitekturen er å redusere koblinga mellom delene av applikasjonen. Det er derfor slik at hvert lag kun kommuniserer med laget over og/eller under. For eksempel: brukergrensesnittlaget kan kommunisere med datamodell-laget og datamodell-laget kan kommunisere med persisteringslaget, men brukergrensesnittlaget kan ikke kommunisere direkte med persisteringslaget og omvendt.

3.6.3 Design



Figur 9 Klassediagram

3.6.4 Datamodell-laget

Datamodell-laget består av fire klasser: Person, Project og XmlSerializer. Disse klassene befinner seg i Java-pakken: no.ntnu.fp.model.

Person- og Project-klassene er dataklassene i kjerneapplikasjonen. Personklassen er en datastruktur som holder informasjonen om personene i adresseboka. Project-klassen er en samling av en eller flere Personobjekter.

XmlSerializer-klassen konverterer data mellom XML-tekst og Java-objekter. Gitt en XML-tekst, konverterer denne klassen til enten Person- eller Gruppeobjekter avhengig av hvilken metode som kalles. Gitt Person- eller Project-klasse konverterer XmlSerializer til XML-tekst.

3.6.5 Persisteringslaget

Persisteringslaget består av et Java-interface, Storage, og en klasse som implementerer dette Java-interfacet, FileStorage. Klassene og interfacet befinner seg i Java-pakken: no.ntnu.fp.storage.

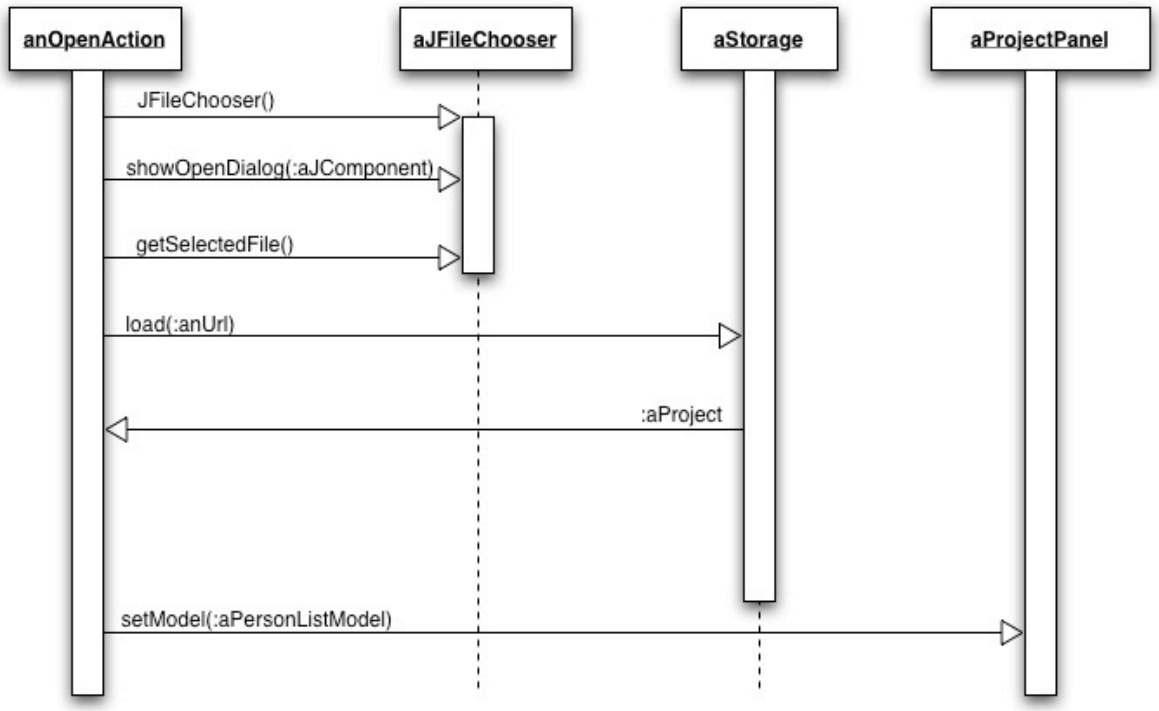
FileStorage-klassen implementerer fellesfunksjonalitet som kreves for å lagre objekter fra datamodell-laget til fil, både til XML-filer og kommaseparert format.

3.6.6 Programsekvenser

Under beskrives de viktigste sekvensene i kjerneapplikasjonen. For leselighetens skyld, er objektene i sekvensflyten gitt det samme navnet som klassen, prefikset av en *a* eller *an*. For eksempel: et objekt av klassen Group omtales i sekvensdiagrammet som aGroup, og et objekt av Storage-klassen omtales i sekvensdiagrammet som aStorage.

3.6.7 Programsekvens: Åpne fil

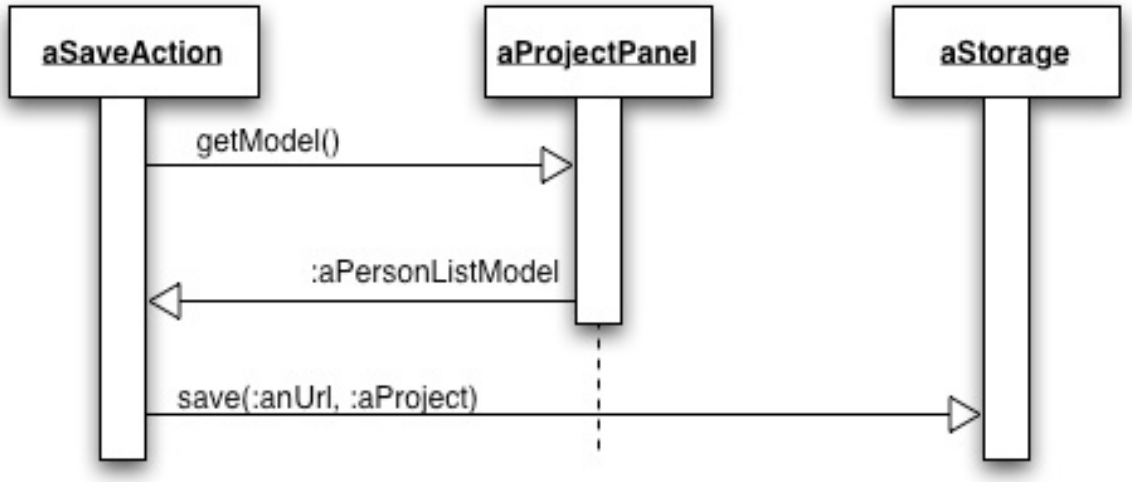
Følgende programsekvens eksekveres når brukeren velger *File -> Open*:



Figur 10 Sekvensdiagram for åpne fil

3.6.8 Programsekvens: Lagre til fil

Følgende programsekvens eksekveres når brukeren velger *File -> Save*:



Figur 11 Programsekvensen for å lagre til fil

3.6.9 Nettverk

Dette kapittelet er ment som en hjelp for å komme i gang med å implementere nettverksstøtte for kjerneapplikasjonen. Kjerneapplikasjonen er tilrettelagt for nettverkskommunikasjon mellom to applikasjoner på hver sin datamaskin. Når du starter kjerneapplikasjonen vil du se et innslag på menylinja som heter *Net*. Om du trekker ned denne menyen har du tre valg: *Connect*, *Disconnect*, *Accept incoming...* Disse tre funksjonene har følgende funksjonalitet:

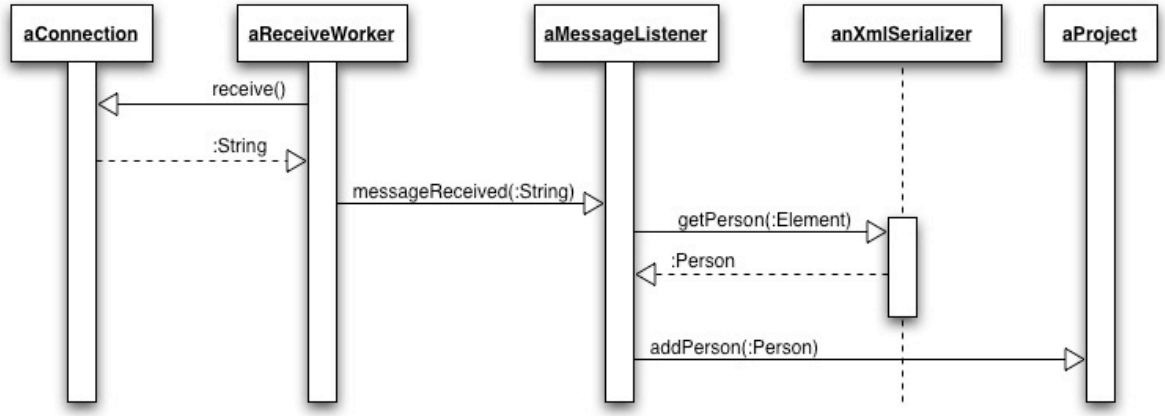
- Med *Accept incoming...* venter applikasjonen på innkommende nettverkskoblinger fra en applikasjon på en annen datamaskin.
- *Connect* kobler applikasjonen opp til en applikasjon som a) kjører på en annen datamaskin, og b) venter på innkommende nettverksoppkoblinger (dvs. brukeren har valgt *Net->Accept incoming...*).
- *Disconnect* bryter oppkobling mot en annen kjerneapplikasjon.

PS: I koden dere har fått utlevert er ingen av disse tre funksjonene implementerte. Om man prøver å kjøre de, får man bare opp en dialogboks som sier *Function not implemented*. Det er fordi oppgaven til nettverksfaget er akkurat det å implementere denne funksjonaliteten.

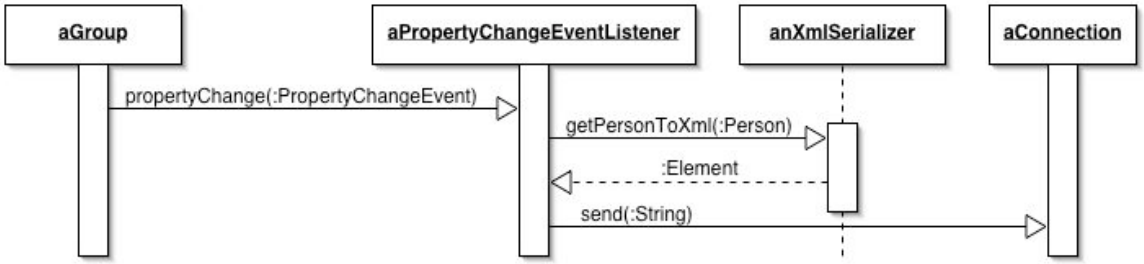
Koden for kjerneapplikasjonen kommer med en pakke som heter no.ntnu.fp.net. Denne pakken inneholder to Java-interface'er–Connection og MessageListener – samt en Java-klasse, ConnectionWorker.

Connection skal implementeres med koden som trengs for å a) koble opp mot en annen maskin, b) vente på oppkoblinger fra en annen maskin, c) motta meldinger fra en annen maskin, og c) sende meldinger til en annen maskin. Dataprotokollen som brukes er det samme XML-formatet som brukes for lagring.

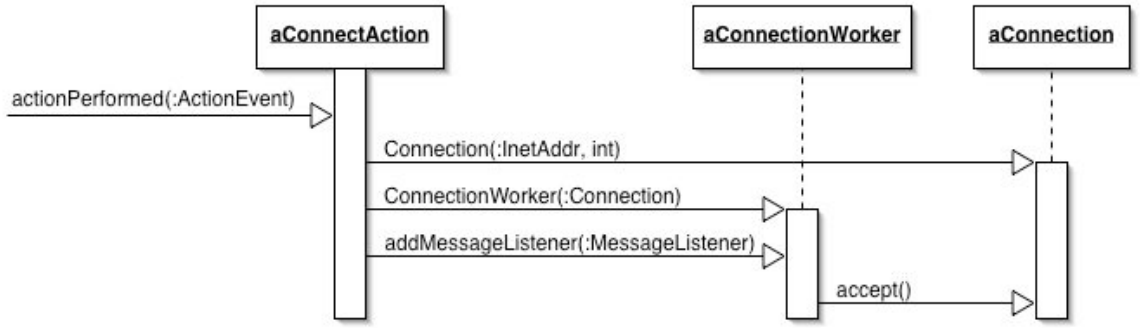
Sekvensen under viser hva som skjer når en kjerneapplikasjon mottar data over nettverket. Merk: dette er kun prinsippene, koden må dere selv implementere. I tillegg til at dere må implementere Connection-interface'et ser dere fra sekvensen over at dere også må implementere MessageListener-interface'et slik at XML-strengen som mottas fra netter blir konvertert til et Personobjekt som kan legges til i et Project-objekt.



Hva så med å sende data? Sekvensen under viser hva dere må implementere for å få det til. Dere må implementere en PropertyChangeListener som lytter på endringer i Group-objektet. Denne PropertyChangeListener-klassen konverterer Personobjektet som er endret/lagt til over til XML ved hjelp av no.ntnu.fp.model.XmlSerializer-klassen. XML-en sendes med send-metoden i Connection-klassen.



Sekvensen under viser stegene for å koble seg opp mot en annen kjerneapplikasjon. Igjen: koden må dere selv skrive. Her bruker vi også ConnectAction-klassen. Denne ligger i no.ntnu.fp.gui-pakken, og inneholder i dag bare testkode for å åpne en dialogboks som forteller deg at funksjonen ikke er implementert. Dere må selv implementere koden i ConnectAction-klassen slik at den kaller de riktige objektene og sette opp applikasjonen slik at den kan sende og motta data fra/til applikasjonen den kobler seg til.



Hver av de tre menyinnslagene *Connect, Disconnect* og *Accept incoming* har korresponderende action-klasser i no.ntnu.fp.gui-pakken: ConnectAction, DisconnectAction og AcceptAction. Som med ConnectAction-klassen, må dere selv implementere den riktige funksjonaliteten i de to andre action-klassene.

4 Trinn og leveranser

Hele prosjektet er oppdelt i tre faser: Planlegging og overordna design, implementasjon og testing og sluttrapportering. Her skal vi gi en overordna beskrivelse av innleveringene som inngår i hver fase i fellesprosjektet. Krav til innleveringer knyttet til hvert enkelt fag finner du på de respektive fagenes hjemmesider eller på "it's learning".

4.1 Fase 1 – Planlegging og overordna design

Det er tre innleveringer tilknyttet til denne fasen som går i uke 11 og 12.

FE1: Prosjektplan

Prosjektplanen beskriver prosjektets mål og midler, blant annet:

- Hvilke ressurser man har til disposisjon, nemlig penger, personell og utstyr.
- Tid og kostnadsestimater for prosjektet
- Hvordan arbeidet skal brytes ned / opp i mindre deler
- Til hvilke tidsfrister hver del skal være ferdig.
- Hvem som skal ha ansvaret for hvilke arbeidsoppgaver.

For mer detaljer om hva planen bør inneholde, samt tips om evt. diagrammer man kan bruke for å få oversikt, se SU-fagets pensum og forelesninger om emnet.

I vårt tilfelle vil nedbrytningen i aktiviteter med tidsfrister langt på vei være gitt av leveranseplanen for prosjektet. Dere bestemmer imidlertid selv når hver enkelt aktivitet skal påbegynnes. Start i god tid, og ha gjerne interne frister underveis mot leveransen, slik at dere kan sjekke innad i gruppen at ting er under kontroll.

Hvem som skal gjøre hva avgjør dere selv, innen visse rammer. Av læringshensyn gjelder følgende krav: *Alle gruppemedlemmer skal være involvert i alle fagdeler*. Deltagelsen i hver fagdel bør betraktes som en investering i et godt eksamensresultat i faget. Det er dessuten neppe lurt å fordele arbeidet slik at en person skal gjøre hele planleggingen, en annen hele nettdelen, osv. Dette vil gi en ubehagelig ujevn arbeidsbelastning gjennom semesteret. I utgangspunktet bør prosjektplanen legge opp til at man skal jobbe jevnt dag for dag med mindre man har spesielle grunner til noe annet. Siden leveransene er av temmelig ulik størrelse, kan en for grov inndeling føre til at noen sitter igjen med "svarteper" mens andre "surfer" i mål med bare enkle arbeidsoppgaver.

Om man har satt opp en prosjektplan med detaljerte personallokering, bør man likevel ikke se på denne som bindende. Det kan være at man finner ut at noe er vanskeligere enn forventet, mens annet er lettere, dvs. at noen har for mye å gjøre og (mens) andre for lite. I så fall bør personallokeringen endres. Dessuten kan man få uforutsett fravær, fordi gruppemedlemmer blir syke eller lignende Dette er det umulig å planlegge på forhånd.

Prosjektplanen må, som et minimum, inneholde følgende kapitler:

- Nedbryting av prosjektet i arbeidspakker – Work Breakdown Structure. Ingen arbeidspakke skal være større enn 16 timeverk.
- Kostnadsoverslag – estimerte timeverk – for hver enkelt pakke. Enkeltoverslagene skal kombineres til et totalestimat for prosjektet.
- Ganttdiagram, som viser hvordan arbeidspakkene er lagt ut i tid og hvilke avhengigheter som finnes mellom arbeidspakkene. Alle leveransene – FE1 – FE4 – skal inn i Ganttdiagrammet.

- Risikoanalysen for prosjektet – hva kan gå galt, hvor sannsynelig er det, hva slags konsekvenser har det og hva kan vi gjøre med det.

FE2: Systemtestplan

Systemtesten er en svarteboks test av hele systemet for å se om det tilfredsstiller de spesifiserte krav. Systemtesten kan ikke utføres før alle systemets deler er implementert og integrert, men denne leveransen gjelder *planen* for testen. Planen skal det være mulig å lage ut fra *brukernes krav* til systemet, dvs. før design og implementasjon. Alle kravene må testes, både de funksjonelle og de ikke-funksjonelle, og planen må angi hvordan dette skal skje. Derfor må hver enkelt test inneholde:

- Formålet med testen
- Hvilken tilstand skal systemet være i
- Hvilke data skal gis inn
- Hvilken respons man forventet fra systemet hvis det fungerer korrekt

Planen må også si noe om hvordan man skal rapportere eventuelle feil som oppdages under testen. Testplanen skal skrives i henhold til standarden IEEE 829.

For mer detaljer om systemtestplaner og deres innhold, se SU-fagets pensum og forelesninger, eventuelt også mulig støttelitteratur.

FE3: Overordna design

Denne leveransen skal inneholde følgende:
- Use case diagrammer for scenariene i kapitel 2.4.1, 2.4.2 og 2.4.3.
- Tekstlige use case for scenariene i kapitel 2.4.2, 2.4.3 og 2.4.4.
- Sekvensdiagrammer for scenariene i kapittel 2.4.1 til 2.4.4
- Beskrivelse av systemets struktur og de viktigste klassene med tilhørende attributter og metoder.

4.2 Fase 2 – Implementasjon og testing

Realisering og testing skal foregå i ukene 15 – 16. Ved slutten av uke 16 skal hver enkelt gruppe levere et system som realiserer alle kravene med de eventuelle forbehold som er gitt i kapittel 4.5.

4.3 Fase 3 – Sluttrapportering

FE4: Sluttrapport, inkludert systemtest og endringsrapport.

Sluttrapporten skal oppsummere erfaringene fra prosjektet og evaluerer hvorvidt det ble vellykket. Hvis man klarte å levere alt det man skulle innen tidsfristene og endte opp med et programsystem som tilfresstilte de angitte kravene, må dette betraktes som vellykket, særlig hvis man også greide å holde seg inne de angitte ressursrammene – dvs. ikke jobbet mer enn de belastningstimer som var forutsatt.

For å kunne rapportere tidsbruken, må dere skrive timer på prosjektet. Hvis man har brukt mange flere timer enn planlagt, bør dette stå i sluttrapporten. Angi også hvilke aktiviteter som først og fremst førte til overskridelsen – eller eventuelt hvilke tekniske eller organisatoriske problemer, hvis dette var årsaken. NB! Man vil ikke få noe kritikk eller større fare for å stryke som følge av tidsoverskridelsene. Dette gjelder også underskridelse, så lenge man leverer det man skal. Rapporter tidsbruken så sannferdig som mulig, disse opplysningene er nyttige for oss for å evaluere opplegget og for eventuelt å gjøre forandringer til neste år.

Sluttrapporten vil inkludere systemtestrapporten, som forteller, punkt for punkt, hvordan systemtesten gikk, dvs. hvilke krav som viste seg å være tilfredsstilt og hva som eventuelt ikke fungerte. Man skal også levere en endringsrapport som inneholder opplysninger om rettinger man har gjort som følge av feil funnet i testfasen. For hver endring må man gjøre nye tester for å forsikre seg om at endringen var en suksess. Disse testene må rapporteres på samme måte som andre tester.

Sluttrapporten skal inneholde følgende kapitler:

- Estimert og virkelig tidsforbruk for hver arbeidspakke. For de arbeidspakkene som avviker mer enn 25 % fra opprinnelig anslag skal det skrives en kort kommentar - ett avsnitt - om de viktigste årsakene til avviket. Rapporter tidsbruken så sannferdig som mulig. Disse opplysningene er nyttige for oss når vi skal evaluere opplegget og eventuelt gjøre forandringer neste år
- Systemtestrapporten, som skal inneholde:
    - Resultatene fra systemtesten, dvs. hvilke krav som ble tilfredsstilt og hva som eventuelt ikke fungerte. Bruk samme nummerering som i kravdelen av dette kompendiet.
    - Endringsrapport, som viser hvilke rettinger man har gjort som følge av feil funnet under systemtesten. For hver endring må man gjøre nye tester for å test at endringen var vellykket. Disse testene må rapporteres på samme måte som andre tester i systemtesten.
- Prosjekterfaringer. Som et minimum må dette kapitlet inneholde et avsnitt om hver av de fire viktigste tingene dere vil legge vekt på å
    - Unngå i senere utviklingsprosjekter.
    - Gjenta i senere utviklingsprosjekter.
    - Forbedre dere for senere utviklingsprosjekter.

5 Innleveringer i TDT4180 Menneske-maskin-interaksjon

Merk: Dette kapitlet gjelder kun for de som tar fellesprosjektet. De som tar faget TDT4180 uten å være del av fellesprosjektet har eget opplegg for øvingene D2 og D3 som finnes på fagets web sider / It's Learning.

5.1 Introduksjon

Design av brukergrensesnitt kan gis en større eller mindre rolle i et utviklingsprosjekt. I TDT4180 er det faglige utgangspunktet at design av brukergrensesnitt i liten grad kan skilles fra spesifikasjon av systemet som helhet. ISO sin definisjon av brukskvalitet (eng: usability) er basert på tre grunnleggende begreper: funksjonalitet (hva systemet lar brukeren gjøre, eng: effectiveness), effektivitet (hvor (ressurs)krevende systemet er i bruk, eng: efficiency) og tilfredsstillelse (hvordan det oppleves å bruke systemet, eng: satisfaction). Brukerens totalopplevelse av et system er en kombinasjon av disse kvalitetsmålene, og dersom en brukergrensesnittdesigner skal stå ansvarlig for denne opplevelsen, er det grunnleggende konseptet for systemet delvis hennes oppgave å utforme, selvsagt i samarbeid med andre. I vårt tilfelle betyr dette at dere som skal lage implementasjonen ideelt sett burde vært med på skrive kravspesifikasjon. Av mange gode grunner er dette ikke tilfelle i fellesprosjektet, så vekten i MMI-delen av prosjektet blir på å gjøre funksjonaliteten lett tilgjengelig for brukeren og å utvikle praktiske ferdigheter i programmering med Java og Swing-rammeverket.

Innleveringene i MMI-faget for de som også tar fellesprosjektet er fordelt på 7 øvinger. De 4 første er teknikkøvinger, dvs. øvinger i Java Swing, og angitt med T1-T4, 2 stk. er designorientert og angitt med D1-D2 og en er både design- og konstruksjonsorientert og angitt D3. Ikke alle øvingene er direkte knyttet til fellesprosjektet, men de er likevel ment å være direkte anvendbare der. Øvingene er oppsummert i tabellen nedenfor.

Navn	Innhold	Fokus
T1	Bruk av knapper, tekstfelt og ulike typer lyttere.	Basisferdigheter og repetisjon fra OO - programmering.
T2	Skjema for innfylling av data og knytning til dataobjekt	Kjennskap til flere basiskomponenter.
T3	Skjema for innfylling forts. JavaBeans og lytting på modellobjekt.	JavaBeans og lytting på modell.
T4	Manipulasjon av liste av objekter vha. aksjoner	Kobling mellom flere komponenter, inkl. JList, JButton m/Action og skjemaet fra T3.
D1	Analytisk evaluering av nettsted.	Evaluering ift. retningslinjer for design
D2	Papirprototyping og evaluering av design i fellesprosjektet.	Papirprototyping og evaluering
D3	Detaljert design og konstruksjon for fellesprosjektet.	Detaljert design og konstruksjon.

Nedenfor presenteres øvingene som er direkte knyttet til fellesprosjektet.

5.2 T1-T4: Basisferdigheter i bruk av Swing og JavaBeans

I øving T1 skal dere lage et enkelt brukergrensesnitt for inntasting av tekst. Øvingen er ment som en repetisjon av Java og GUI-programmering fra programmeringskurset.

I T2 skal dere implementere en enkel modellklasse og lage et innfyllingsskjema for dennes egenskaper. Et tilsvarende skjema er relevant for både fellesprosjektapplikasjonen. T2 dekker bruk av en del enkle GUI-elementer, inkludert hvordan disse lar brukeren se (view) og endre (control) enkle verdier. I T2 vil bruk av GUI-elementene gjøre at verdier forplantes fra GUI til modellobjekter. Dersom brukeren f.eks. endrer på teksten i et tekstfelt, skal navnegenskapen til et personobjekt endres tilsvarende.

I T3 skal skjemaet i T2 utvides med JavaBean-hendelser slik at endringer i modellobjekter forplantes tilbake. Dersom f.eks. navnegenskapen til et personobjekt endres, skal tekstfeltet oppdateres automatisk. Dette er viktig dersom modellobjekter endres av andre deler av applikasjonen, f.eks. på bakgrunn av nettverkskommunikasjon.

I T4-øvingen skal skjemaet i T3 knyttes til en liste hvor en kan velge hvilket objekt som skjemaet skal inneholde. En skal dessuten kunne lage og legge nye modellobjekter til lista og fjerne elementer fra lista.

For mer spesifikke krav til T2-T4 og detaljer rundt gjennomføring og innlevering, se MMI-fagets nettsider/It's Learning.

5.3 D2: Brukbarhetstest av papirprototyp

Denne innleveringen skal inneholde en såkalt *papirprototyp* av deler av brukergrensesnittet til fellesprosjektapplikasjonen. Bruk scenariene 1-4 i kap. 3.3. Papirprototypen skal brukbarhetstestes med en annen gruppe i prosjektet. Innleveringen skal inneholde en testplan, en beskrivelse av funn, og om nødvendig forslag til re-design. Redesignet skal ligge til grunn for det som skal beskrives nærmere og konstrueres i D3-øvingen og til sist realiseres innenfor fellesprosjektet. Metoder for å lage papirprototyper og gjøre såkalte "Wizard of Oz"-brukbarhetstester vil bli gjennomgått i MMI-faget (se forøvrig vedlegg).

5.4 D3: Skjermdesign og konstruksjon av brukergrensesnittet

Denne innleveringen består av to deler:

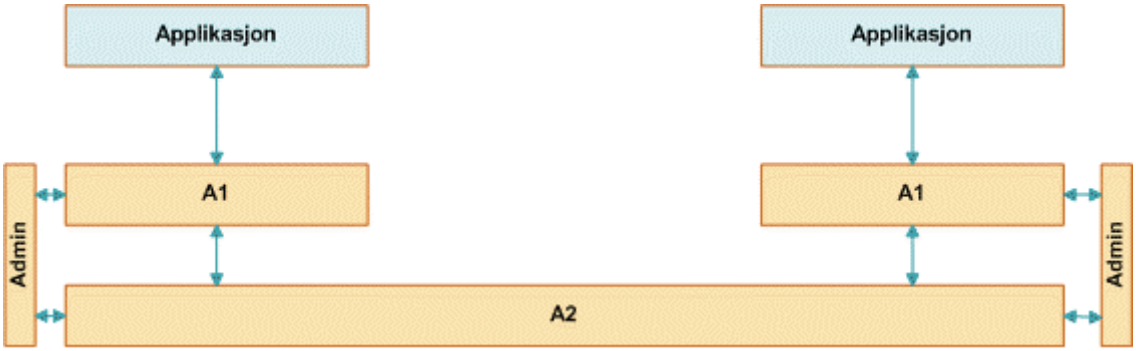
- **Skjermdesign**. Dere skal her ta utgangspunkt i kravspesifikasjonen og scenariene 1-4 som er beskrevet i kap. 3.3. I innleveringen skal dere utdype grafisk struktur og utforming, og spesifisere hvordan alle deler av applikasjonen reagerer på relevante hendelser, som museklikk og tastetrykk. Målet er å spesifisere *brukeropplevelsen*, dvs. hva brukeren til enhver tid ser og kan gjøre.

- **Konstruksjonsbeskrivelse**. Mens skjermdesignet fokuserer på brukerens opplevelse, skal dere her beskrive hvordan brukergrensesnittet er bygd opp, dvs. hvilke vinduer og dialogelementer som utgjør grensesnittet, og hvordan disse er koblet sammen i et hierarki og kommuniserer vha. metodekall og hendelser. Her er altså fokuset hvordan konstruksjonselementene fra Swing-rammeverket benyttes for å realisere brukergrensesnittet. Detaljeringsnivået skal tilsvare formuleringen av konstruksjonsøvingene og i prinsippet gjøre det mulig å skrive en funksjonstest vha. JUnit/JFCUnit-rammeverket. Teknikker og notasjon for å beskrive oppbygningen baseres på UML og vil bli forelest.

6 TTM4100: Kommunikasjon, tjenester og nett (KTN)

Applikasjonen som utvikles i fellesprosjektet er avhengig av å kunne kommunisere over et nettverk. Målet med oppgaven i KTN er å kunne tilby denne applikasjonen en pålitelig overføring av informasjon.

Et proprietært forbindelsesløst nett, kalt *A2*, eksisterer allerede. Det er mulig å kontrollere feilene som kan oppstå i A2 ved å bruke *Admin*. Vi skal realisere forbindelsesorientert nett basert på A2. Dette forbindelsesorienterte laget er kalt *A1*, og er kun delvis implementert. Deres oppgave blir derfor å gjøre de nødvendige endringer i A1, slik at vi kan tilby pålitelig overføring av informasjon til applikasjonen. Figuren under skisserer hvordan komponentene forholder seg til hverandre.



Både A2 og Admin er ferdig implementert og skal ikke endres. Dokumentasjon av A2 er gitt i [A2-Ud] og dokumentasjon for Admin er gitt i [Adm-Ud]. Disse blir levert i egne javapakker kalt henholdsvis `no.ntnu.fp.net.cl` (A2) og `no.ntnu.fp.net.admin` (Admin).

Alle forandringer som er nødvendig for å tilby forbindelsesorientert overføring for applikasjonen gjøres i A1. Et koderammeverk [A1-Djc] med tilhørende dokumentasjon [A1-Doc] som kan benyttes som basis blir utlevert. Som et utgangspunkt for design og realisering av denne eksisterer en dokumentasjon [A1-Ud] som kan benyttes som et kravgrunnlag for realisering av A1.

6.1 Dokumentasjon

I tilknytning til oppgaven finnes det en del dokumentasjon som vil være til hjelp for å forstå oppgaven og de komponenter som benyttes i realiseringen. Dokumentene vil være tilgjengelige hjemmesiden til KTN og det anbefales å studere disse nøye. Sjekk alltid dokumentasjonen først hvis dere står fast! De ulike dokumentene er:

[A1-Ud] – Brukerdokumentasjon for A1 (User documentation)
[A1-Doc] – Dokumentasjon av den distribuerte koden i A1 (Documentation)
[A1-Djc] – Den distribuerte koden i A1 (Distributed java code)
[A2-Ud] – Brukerdokumentasjon for A2 (User documentation)
[Adm-Ud] – Brukerdokumentasjon for Admin (User documentation)

6.2 Funksjonelle krav

Dette avsnittet presenterer de funksjonelle kravene som må oppfylles for å kunne få innleveringene godkjent.

6.2.1 Grensesnitt: Applikasjon / A1 og A1 / A2

1. Applikasjonen skal benytte seg av grensesnittet gitt i `Connection`-klassen for å kommunisere over nettet dere skal tilby. Se [A1-Ud].

2. A1 skal kommunisere med A2 gjennom de tre metodene `send()`, `receive()` og `cancel_receive()`. Se [A2-Ud].

6.2.2 Krav til tilkobling

3. A1 skal tilby brukeren en forbindelsesorientert tilkobling mellom to instanser av A1.

4. Alle feil som oppstår i det underliggende A2 skal tas hånd i A1, og skal være transparente (usynlige) for applikasjonen. Se [A2-Ud] for en oversikt over hvilke feil som kan oppstå i A2. Kun om feilene er så alvorlige at forbindelsen er/kan regnes som brutt skal applikasjonen oppdage disse.

5. A1 skal kunne utføre følgende handlinger i sin interaksjon med en annen instans:
   a. Koble til
   b. Koble fra
   c. Motta data
   d. Sende data

6.3 Innleveringer

Innleveringene i KTN skjer uavhengig av de andre delene i fellesprosjektet. Dette betyr at dere ikke skal levere hele applikasjonen eller all dokumentasjonen til oss, men kun de delene av prosjektet som omhandler KTN-delen av prosjektet.

KTN1 omhandler design av løsningen og skal utføres i de to første ukene av prosjektarbeidet. I KTN2 skal dere implementere selve løsningen og dette gjøres i de tre resterende ukene.

6.3.1 KTN1

Beskrivelse:

Dere skal her vise hva dere tenker å gjøre med A1 for å kunne løse oppgaven. KTN1 skal leveres i to faser. Fase 1 er en foreløpig innlevering slik at dere kan få tilbakemelding på hvordan dere tenker å løse oppgaven. NB: Selv om den er foreløpig, må alle levere den! Basert på tilbakemeldingen dere får på fase 1, kan dere forbedre og gjøre de nødvendige endringene før dere leverer fase 2 en uke senere. Ved denne faseinndelingen kan dere få en pekepinn på hva dere trenger å forandre for at fase 2, og dermed KTN1, kan bli godkjent.

Mål:

Dere skal vise at dere har forstått oppgaven, samt designe og beskrive hva dere har tenkt til å gjøre i implementeringsfasen (KTN2). Dere skal vise hvordan dere vil oppfylle de funksjonelle kravene for å kunne tilby forbindelsesorientert overføring til applikasjonen.

Hva?

- Sekvensdiagrammer for samspillet mellom Applikasjonen, A1 og A2. Diagrammene må klart vise hvordan kravene i 6.2 skal oppfylles. Scenarioer for både oppkobling, overføring av data og nedkobling bør vises.
- Tilstandsdiagrammer for A1.
- En overordnet tekstlig beskrivelse av design og realisering av A1, og av totalløsningen, inklusive bruk av komponentene Admin og A2.
- En tekstlig beskrivelse av hvordan dere tenker å oppfylle krav 4 fra 6.2 (feilhåndtering). Fremgangsmåten må diskuteres og begrunnes. Det skal gå klart fram hvordan hver feiltype blir håndtert.
- Testplan for testing av ferdig implementert A1, der dere klart viser hvordan dere systematisk skal teste at alle kravene i 6.2 er oppfylt. Dere skal som et minimum teste følgende:
  - Systemet skal testes uten feil.
  - Hver feilkategori skal testes isolert ved feilsannsynlighet på 10 % og 50 %
  - Tester skal også gjennomføres der flere testkategorier er representert samtidig. Dere bestemmer selv hvor mange, hvilke kombinasjoner og hvilken feilsannsynlighet som skal benyttes for hver av testene. Valgene skal dokumenteres og begrunnes.
  - Tester der alle feilkategoriene er representert. Dere velger selv hvilken feilsannsynlighet hver av kategoriene skal ha. Valgene skal dokumenteres og begrunnes.

Hvordan?

Materialet leveres gruppevis gjennom It`s Learning.

Når?

Innleveringsfrist KTN1 fase 1: **Se it's learning**
Innleveringsfrist KTN1 fase 2: **Se it's learning**

6.3.2 KTN2

Beskrivelse:

Basert på det arbeid som ble gjort i KTN1, skal dere nå implementere løsningen. Dere skal gjøre de nødvendige endringene og forbedringene i A1, slik at feilene som kan oppstå i A2 ikke merkes av applikasjonen.

Mål:

Applikasjonen skal kunne kommunisere over nettverket ved hjelp av A1. A1 skal håndtere alle feil som oppstår i underliggende nettverk og sørge for at applikasjonen oppfatter nettverket som forbindelsesorientert og pålitelig.

Hva?

- Oppdaterte diagrammer og beskrivelser fra KTN1 hvis endringer er gjort. Dere skal også forklare kort hvorfor endringene ble gjort. Det er et absolutt krav at det er en overensstemmelse mellom dokumentasjonen (diagrammene og beskrivelsen) og den implementasjonen dere gjør.
- Alt dere har produsert av kode skal leveres. Javadoc kreves ikke, men kommentarer må være med der det er nødvendig for å forstå hva dere har gjort. Tilstreb å lage oversiktlig og forståelig kode, da vi må skjønne hva dere har gjort for å kunne godkjenne arbeidet.
- Testlogg (utfylt testplan fra KTN1) for gjennomførte tester av A1. Eventuelle avvik fra ønsket resultat må kommenteres og diskuteres.
- Demonstrasjon for studentassistenten med alle gruppemedlemmer til stede.

Hvordan?

Materialet leveres gruppevis gjennom It`s Learning. Demonstrasjonen kan skje etter avtale når dere mener dere er ferdige.

Når?

Innleveringsfrist KTN2: **ZZ**

7 TDT4145 Datamodellering og databasesystemer

7.1 Databasedelen av fellesprosjektet

DB1: Konseptuel datamodell *Innlevering: Se it's learning*

Dere skal lage en konseptuel databasemodell som oppfyller alle punkter **i kravspesifikasjonen som er gitt i fellesprosjektheftet**. Det er to ting som skal leveres:

- Et diagram som viser datamodellen. Det er valgfritt om dere vil bruke UML eller EER, og om dere vil tegne for hånd eller bruke modelleringsverktøy. Ta med alle entitetsklasser, relasjonsklasser, attributter, kardinaliteter, eventuelle eksistensavhengigheter og svake entitetsklasser.
- Et dokument som beskriver hvordan modellen oppfyller kravspesifikasjonen. For hvert nummererte krav i kravspesifikasjonen skal det kort (et par linjer burde holde) forklares hvordan modellen deres oppfyller kravet.

For å bli godkjent må modellen deres oppfylle absolutt alle krav i kravspesifikasjonen. Kravoppfyllelsesdokumentet er en forutsetning for at vi skal kunne sjekke dette noenlunde enkelt, så innleveringer uten dette vil ikke bli godkjent.

Besvarelsen skal leveres samlet i PDF. Husk å skrive gruppenummer og navn på alle i gruppa på innleveringen.

DB2: Logisk databaseskjema *Innlevering: Se it's learning*

Den konseptuelle modellen fra Oppgave 1 skal omformes til et logisk databaseskjema, i form av et SQL-skript som skal kunne brukes til å generere en relasjonsdatabase i MySQL. Det er to ting som skal leveres:

- Et kjørbart SQL-skript som genererer databasen. Husk alle primær- og fremmednøkler, og eventuelt andre restriksjoner (constraints).
- Oppgave 1-innleveringen. Hvis dere har endret modellen siden oppgave 1, lag en revidert versjon av modellen. Dere står fritt til å endre datamodellen til enhver tid, men alle krav må alltid være oppfylt. Husk å oppdatere kravoppfyllelsedokumentet, og marker endringene slik at vi slipper å sjekke alt på nytt. Vitsen med å levere inn oppgave 1 igjen, er at vi skal kunne kontrollere at dere har laget SQL-koden korrekt i henhold til modellen. Det må derfor være samsvar mellom modellen og SQL-koden for at innleveringen skal bli godkjent.

SQL-scriptet skal lastes opp som en separat tekstfil og resten skal samles i PDF. Husk å skrive gruppenummer og navn på alle i gruppa på innleveringen.

8 Vedlegg A - En rask introduksjon til UML

Guttorm Sindre, desember/januar 2000

Formålet med dette notatet er å gi en rask introduksjon til UML slik det skal brukes i faget Systemutvikling og prosjektarbeidet i forbindelse med dette. Følgelig går vi ikke inn på alle deler eller detaljer ved UML, bare de som er mest relevante for faget. De som er interessert i å vite mer om UML, kan se den støttelitteraturen som er referert på slutten av dette notatet. Merk også at modelleringsmulighetene vil variere noe avhengig av hvilket verktøy man bruker. Dette notatet inneholder ingen diskusjon av verktøy, tar kun for seg UML-modellering på et konseptuelt plan.

8.1 UML – hva og hvorfor?

Hvorfor det er viktig med analyse og design før man programmerer, og hvorfor det er viktig å bruke modeller i disse fasene, vil være diskutert i læreboken – likeledes fordeler med diagrammer i forhold til rent tekstlige beskrivelser. I læreboken er det også forklart at det fins flere paradigmer for modellering (funksjonsorientert, objektorientert…) – som hver kan ha sine fordeler og ulemper. Denne diskusjonen lar vi derfor ligge her. Vi vil også anta at sentrale begreper i objektorientert programmering, slik som klasse, metode, arv, grensesnitt, etc. er kjent fra faget SIF8005 eller tilsvarende forkunnskaper, slik at det ikke er nødvendig å forklare disse her.

UML (Unified Modelling Language) er et objektorientert modelleringsspråk, ment brukt i analyse og design. Rundt 1990 oppsto det en rekke slike språk, med større eller mindre variasjon i konsepter og notasjon. Man så snart et behov for standardisering. Grady Booch, Ivar Jacobson og James Rumbaugh, som tidligere hadde laget hvert sitt modelleringsspråk, bestemte seg for å stikke hodene sammen, og kom opp med UML. Dette er nå blitt en slags de facto industristandard, anbefalt av OMG (Object Management Group, en organisasjon som arbeider for standardisering av objektorienterte språk og teknologier).

UML kan sies å bestå av en rekke *subspråk* – ulike typer diagrammer som brukes til hver sine deloppgaver i modelleringen. Hvert av disse subspråkene inkluderer visse

- **konsepter.** Akkurat som naturlige språk som norsk har ordklasser (substantiv, adjektiv, verb, etc.), vil modelleringsspråk ha ulike typer konstruksjoner som settes sammen for å lage beskrivelser av et system. Språkets *syntaks* (eller om man vil: grammatikk) regulerer hvordan konsepter kan settes sammen til mer komplekse uttrykk, og språkets *semantikk* sier noe om hva disse uttrykkene betyr. Innen hvert språk vil man dessuten ha en viss
- **notasjon,** dvs. en bestemt måte å skrive/vise konseptene på, f.eks. i et diagram. I mange tilfeller kan konsept og notasjon virke som to sider av samme sak, men rent formelt er det greit å skille mellom disse. Man kan godt ha flere alternative notasjoner for samme konsept. For ER-diagrammer fins det f.eks. en variant hvor attributter angis som rundinger tilknyttet entitetsklassers firkanter, og en annen variant hvor attributtene simpelthen navngis inni entitetsklassenes firkanter. Man har også eksempler på notasjon som kan bety ulike ting i ulike sammenhenger, slik som rektangler og romber – i algoritmiske flytdiagrammer betyr dette handlinger og valg, mens det i ER-diagrammer vil bety entitets- og relasjonsklasser.

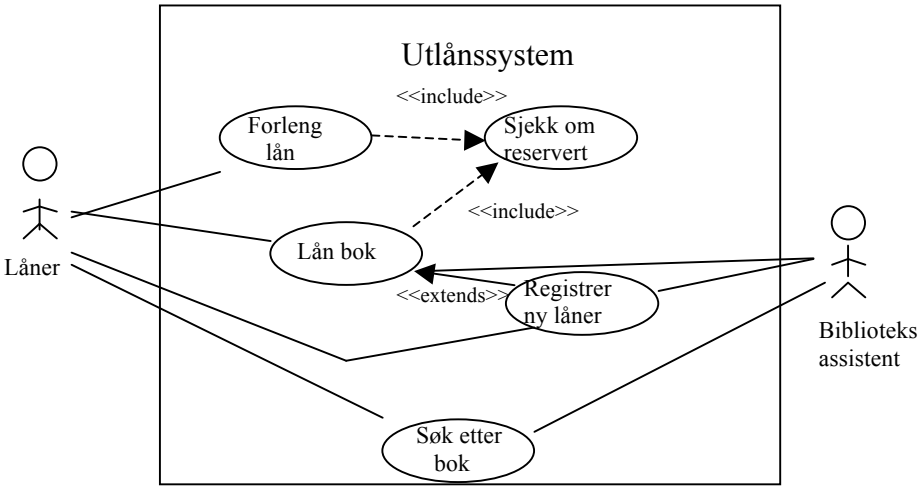
I det følgende skal vi ta for oss disse subspråkene innen UML:

- brukstilfellediagrammer (use case diagrams)

- klassediagrammer (class diagrams)
- interaksjonsdiagrammer
- tilstandsdiagrammer (state transition diagrams)

8.2 Brukstilfellediagrammer

UML inneholder ikke språk for å skrive noe som kan kalles kravspesifikasjoner. En viss modellering av brukerens interaksjon med systemet er likevel mulig, i form av brukstilfellediagrammer. Disse kan benyttes enten som en hjelp til å komme fram til en kravspesifikasjon, eller for å få bedre oversikt og forståelse for en kravspesifikasjon som allerede foreligger. Det diagrammene viser, er først og fremst hvilke funksjoner systemet skal ha, og hvilke aktører som trenger de ulike funksjonene.



Figur 1: Brukstilfellediagram

Figur 1 viser et brukstilfellediagram for et biblioteksystem hvor kundene enten kan registrere sine lån selv via en automat, eller på mer tradisjonelt vis ved å henvende seg til en assistent. Dette er ikke et fullstendig brukstilfellediagram for systemet, man kunne også tenke seg mange andre funksjoner, f.eks. innlevering, purring, tapsmelding, erstatningskrav, …, og flere roller, f.eks. bibliotekar, som i motsetning til underordnede assistenter vil kunne foreta bestilling av nye bøker o.l. I det følgende vil vi diskutere de sentrale konseptene som der benyttet i brukstilfellediagrammet.

8.2.1 Aktør

Aktører blir vist som menneskelignende strektegninger og skal forestille mulige brukere av systemet, med et forklarende navn (”Låner” og ”Biblioteksassistent” i diagrammet). Det er viktig å merke seg at aktøren ikke representerer en bestemt person, men en *rolle* overfor systemet. Flere ulike personer kan tenkes å spille samme rolle (f.eks. kan biblioteket ha 10 assistenter og tusenvis av lånere), og en og samme person kan ha ulike roller vis à vis systemet på ulike tidspunkt. En biblioteksassistent kan selv låne bøker og dermed opptre i en lånerrolle. I eksemplet er aktørene likevel enkeltpersoner (for hver enkelt interaksjon), men dette trenger ikke alltid å være tilfelle: det kan også være grupper av personer eller hele etater. Hvis systemet hadde en funksjon ”Generer årsrapport” og årsrapporten bl.a. skulle sendes til kommunen, kunne man hatt ”Kommune” som en aktør i diagrammet. I noen tilfeller kan også programsystemer figurere som aktører. Hvis biblioteket fra før av hadde et regnskapssystem og det nye utlånssystemet skulle utveksle opplysninger med dette, kunne funksjoner som

innkjøp av bøker, meldinger om tap og innkreving av gebyrer ha ”Regnskapssystem” som en tilknyttet aktør.

8.2.2 System

Systemet vises som et rektangel. Dette skal forestille det automatiserte informasjonssystemet, hvis navn vil være skrevet øverst i rektanglet (her ”Utlånssystem”). Diagrammene kan brukes både til å modellere den eksisterende situasjonen eller en ønsket fremtidssituasjon. I forbindelse med systemutvikling er ofte det siste det mest interessante, rektanglet vil i så fall indikere et system som foreløpig ikke eksisterer, men som man har tenkt å lage. Hvis man i en innledende fase er usikker på hvordan automatiseringsgrensen skal trekkes (dvs. hva som kan utføres automatisk og hva som må gjøres manuelt av mennesker), kan man starte modelleringen uten å ha noe systemrektangel og prøve å plassere dette etter hvert. Det kan også tenkes at man ønsker å modellere menneskers manuelle aktiviteter, dette må i så fall plasseres utenfor systemboksen (dog ikke særlig vanlig i UML-sammenheng).

8.2.3 Brukstilfelle

Brukstilfeller (use cases) vises som navngitte oval. Vårt diagram inneholder fem ulike brukstilfeller. Et brukstilfelle kan sies å være ett tilfelle/eksempel på hvordan systemet kan brukes, én av mange funksjoner som brukerne skal tilbys. Brukstilfeller kan variere i kompleksitet. For tekstbehandling vil f.eks. ett brukstilfelle være å slå på kursivskrift, et annet å gjøre rettskrivingskontroll på hele dokumentet. Her er det første en ytterst banal operasjon, mens det siste er langt mer komplekst og kan kreve videre interaksjon med brukeren underveis, f.eks. spørsmål om hva man skal gjøre hvis man finner feil. Et vanlig råd er at man ikke bør henge seg opp i for små og detaljerte brukstilfeller tidlig i modelleringen, men starte med de store og viktige først.

8.2.4 Deltagelse

Deltagelse er en relasjon mellom en aktør og et brukstilfelle, vist ved en vanlig linje mellom disse to. Linjen har ingen tekst tilknyttet seg. Betydningen er at denne aktøren er involvert i dette brukstilfellet på en eller annen måte. Det kan være at aktøren tar initiativ til brukstilfellet, altså er den som direkte ber systemet om å utføre en viss funksjon. Men koblingen kan også være løsere, f.eks. at aktøren på en eller annen måte bidrar med input eller er interessert i output fra brukstilfellet, eller på annen måte berøres av at funksjonen blir utført.

8.2.5 Bruk

Dette er rettede relasjoner mellom to brukstilfeller, vist ved piler. Betydningen er at en funksjon i systemet benytter seg av en annen funksjon. Det fins to ulike bruksrelasjoner:

- *Ubetinget bruk*, som innebærer at den ene funksjonen alltid vil komme til å kalle den andre. I dette tilfellet er ordet *uses* tilknyttet pilen, som går fra den kallende funksjonen til den kallede. Denne relasjonen er nyttig hvis to eller flere funksjoner har samme subfunksjonalitet, denne kan da skilles ut som et eget brukstilfelle som de andre kan knyttes til ved en *uses*-relasjon. I vårt eksempel ser vi at både ”Lån bok” og ”Forleng lån” benytter seg av ”Sjekk om reservert” – hvis en annen låner har reservert boken, kan forespørselen ikke tilfredsstilles.
- *Betinget bruk,* som innebærer at den ene funksjonen av og til vil komme til å kalle den andre, f.eks. hvis noe går galt, slik at man må gjøre noe utover standard prosessering. I dette tilfellet er ordet *extends* tilknyttet pilen, som nå går fra den kallede funksjonen til den kallende. I vårt eksempel er dette tilfellet for ”Registrer ny låner”, som altså utvider ”Lån bok”. Normalt vil funksjonen ”Lån bok” ikke ha noe behov for ”Registrer ny låner”.

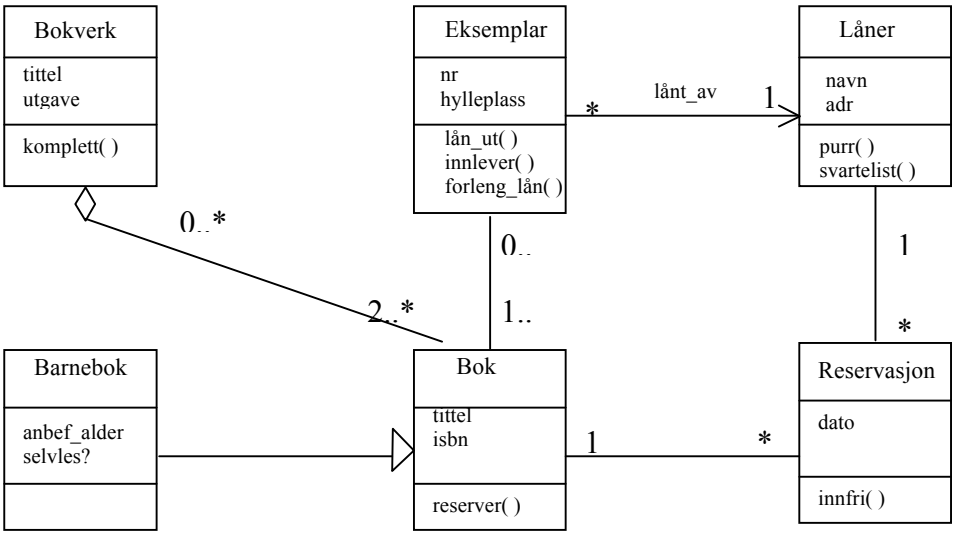
Men hvis en person kommer og vil låne en bok, og *ikke* er registrert som låner ennå, må dette gjøres før lånet kan gjennomføres.

At pilen går motsatt veg i to såpass beslektede situasjoner, er kanskje litt uheldig, men det er nå slik språket er. Hvis man ser det fra brukerens synsvinkel, kan det likevel være fornuftig. I tilfellet "uses" er subfunksjonen alene kanskje ikke tilgjengelig eller interessant for brukeren som sådan. I diagrammet er det ingen aktør som bruker "Sjekk om reservert" direkte, uten foranledning. Dette gjøres kun når noen er interessert i å sikre seg boken, og kan således oppfattes som en subfunksjon som ikke har interesse uavhengig av andre oppgaver. Mhp. "extends" vil den kallede funksjonen gjerne være interessant på selvstendig basis. Vanligvis vil man jo gjøre "Registrer ny låner" uten at man først prøver å låne en bok.

8.3 Klassediagrammer

Klassediagrammer er en sentral teknikk i objektorientert analyse (OOA) og design (OOD). Man kan imidlertid merke seg at begrepet OOA ikke innebærer det som enkelte andre kaller analyse (nemlig å undersøke brukernes behov og lage en kravspesifikasjon basert på dette), men å finne ut – etter at en slags kravspesifikasjon foreligger – hvilke objekter som trengs, eventuelt å få større klarhet i kravspesifikasjonen ved å undersøke hvordan den kan uttrykkes i en objektorientert modell. Kunden etterspør jo normalt ikke objekter eller klasser, men et system som tar visse typer input og leverer visse typer output innen rimelig tid, med et kurant brukergrensesnitt. For kunden spiller det gjerne ingen rolle om systemet er modellert og realisert i henhold til et objektorientert, proseduralt eller funksjonelt paradigme, så lenge det gjør jobben på en tilfredsstillende måte.

Figur 2 viser (deler av) et klassediagram for utlånssystemet diskutert tidligere. Vi vil nå forklare de sentrale konseptene basert på dette.



Figur 2: Klassediagram

8.3.1 Klasse

Klasser vises som rektangler, med klassens navn øverst. I tillegg til navnet kan rektanglet inneholde *attributter* og *metoder* for klassen, i så fall er det vanlig å skille ut disse med horisontale delelinjer i rektanglet. Hvis man verken ønsker å oppgi attributter eller metoder, kan man droppe delelinjene. Siden en klasse kan ha et stort antall attributter og metoder, er det ofte uhensiktsmessig å nevne alle i diagrammet, i all fall på et tidlig stadium i designet, der det viktigste er å få oversikt.

De seks klassene i vårt eksempel skulle være nokså selvforklarende. Man kan merke seg forskjellen på klassene Bok, som vil være de logiske bøkene (f.eks. ”Sult” av Hamsun, ”L” av Erlend Loe) og Eksemplar, som vil være ett (av muligens flere) fysiske eksemplarer biblioteket har av boken. Det er dermed ikke bøker men *eksemplarer* som de facto kan lånes ut til lånere og leveres inn igjen av disse. Derimot synes det mest fornuftig at klassen Bok har metoden *reserver( )*. Hvis en låner ønsker å reservere f.eks. ”L”, og alle bibliotekets eksemplarer er utlånt, ønsker man normalt å få kloa i det (ukjente) eksemplaret som viser seg å bli først innlevert, ikke på forhånd å knytte reservasjonen til ett bestemt eksemplar (som kanskje aldri blir innlevert). Hvis biblioteket kun hadde ett eksemplar av hver bok, ville derimot skillet mellom Bok og Eksemplar ha vært unødvendig.

Klassen Bokverk brukes kun for verk som består av flere bind, og er utstyrt med en metode *komplett( )*, for å kunne sjekke om verket er komplett (dvs. biblioteket har minst ett eksemplar av alle bøker som inngår i bindet). Låner har en metode *svarteliste( )*, denne benyttes hvis låneren har mer enn et visst beløp utestående i gebyrer, eller har mange bøker som ikke er innlevert i tide, eller ved andre graverende forhold (f.eks. upassende oppførsel i bibliotekets lokaler). Låneren vil da ikke få låne flere bøker før det er ordnet opp i forholdet som førte til svartelistingen.

Vårt diagram inneholder langt ifra alle attributter og metoder som ville være aktuelle, plasshensyn forhindrer oss fra å ta med alt. F.eks. måtte Eksemplar ha opplysninger om innkjøpsår og pris, når det sist ble utlånt, når det sist ble innlevert, når det sist ble vedlikeholdt. Mange av våre utelatelser er mest av plasshensyn og latskap. Men en del attributter og metoder kan det uansett være lurt å la være å nevne, i all fall på et tidlig stadium i modelleringen, der oversikt er det viktigste. Dette gjelder særlig konstruktorer, samt get- og set-metoder for nevnte attributter. Hvis man har til hensikt å lage god objektorientert design, er nemlig innlysende at slike metoder vil finnes. Når det gjelder attributter, kan man la være å nevne de som er implisitte gjennom tilbudte metoder. Siden Låner har metoden *svarteliste( )*, synes det unødvendig også å liste opp en attributtstatus, som kan være enten OK eller svartelistet. Attributter som er implisitte som følge av de viste relasjonene (se kommentar nedenfor), kan også gjerne utelates, i all fall på et tidlig stadium.

8.3.2 Relasjoner

Relasjoner mellom klassene uttrykker sammenhenger mellom disse. Det fins tre hovedtyper slike relasjoner:

- Vanlige (assosiative) relasjoner
- Generalisering
- Aggregering

I det følgende vil vi ta for oss hver av disse i mer detalj.

A) Vanlige (assosiative) relasjoner. Disse vises ved enkle linjer, uten noen spesiell symbolbruk, annet enn eventuelt vanlige strekpiler i enden for å vise navigasjonsretningen. I vårt eksempel har vi stort sett latt vær å angi navigasjonsretningen, den er kun med for forholdet mellom ”Eksemplar” og ”Låner”, pilen indikerer her at vi fra et eksemplarobjekt direkte vil kunne finne det tilhørende lånerobjektet, men ikke omvendt. Implementasjonsmessig vil dette bety at eksemplaret har en attributt med referanse til et lånerobjekt. Vi kunne også ha valgt det motsatte, nemlig at hvert lånerobjekt inneholdt en liste med referanser til de eksemplarobjekter låneren p.t. hadde i sin besittelse. Eller man kunne gjort begge deler, og dermed hatt muligheten til å navigere begge veier (fordel med hensyn på raskere navigering, ulempe fordi man får redundans i datastrukturen, noe som vil gi

langsommere oppdatering og større fare for feil). Hvilken variant man velger vil avhenge av hva slags funksjonalitet brukerne vanligvis etterspør, og med hvilke effektivitetsbehov. Normalt vil det lønne seg å utsette inngående vurderinger av navigasjonsretning til man har fått klarhet i hvilke klasser og relasjoner man skal ha. Enkel linje (uten noen piler) betyr altså at navigasjonsretningen foreløpig ikke er bestemt, mens dobbel navigasjonsretning vil bli vist ved at man har pil begge veier.

Hvis det er tvil om betydningen av en relasjon, bør den også navngis. I vårt eksempel anses relasjonene for selvforklarende, men vi har likevel vist navngiving mellom Eksemplar og Låner. Nær endene av hver relasjonslinje vil man finne opplysninger om relasjonens *kardinalitet*, à la hva man har for ER-diagrammer. Man kan for det første angi om en relasjon er til-en (1) eller til-mange (*). Hvis man ønsker det, kan man også angi om deltagelse i relasjonen er obligatorisk eller frivillig (1..1 eller 1..* vil signalisere obligatorisk deltagelse, mens 0..1 eller 0..* signaliserer frivillig deltagelse). Frivillig/obligatorisk kan anses som et mer detaljert spørsmål enn en/mange, man bør derfor få klarhet i det siste først.

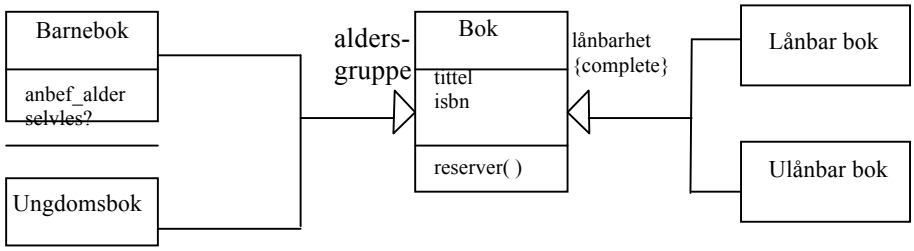
Vi ser fra diagrammet at en låner kan ha lånt flere eksemplarer (*), men hvert eksemplar vil kun være lånt av en låner av gangen (1). Et eksemplar vil kun være eksemplar av én bok (1..1), og det må også være eksemplar av en bok (1..1). Derimot kan bøker finnes i mange eksemplarer (0:*), og det kan finnes bøker vi ikke har noen eksemplarer av (0:*). I dette siste tilfellet kan det kanskje synes rart at man fortsatt ønsker å ha opplysninger om boken lagret i systemet, men dette kan være interessant i noen situasjoner, f.eks. hvis det er en bok som er blitt mistet og som var del av et større bokverk, hvilket vil tilsi et sterkt incentiv for å prøve å skaffe seg boken igjen, og dermed komplettere bokverket.

B) Generalisering. Mange kaller denne relasjonen for arv eller arving. Navnet på selve den konseptuelle relasjonen er imidlertid generalisering, eventuelt den inverse *spesialisering*, mens arving mer er en mekanisme i objektorienterte programmeringsspråk. Dvs., man kan godt tenke seg modelleringsspråk hvor man kan uttrykke generalisering, uten at det dermed fins en arvemekanisme. I mengdealgebraisk forstand kan generalisering sies å tilsvare en overmengderelasjon (*superset*). Hvis vi alltid har $A \supset B$, er A en generalisering av B, dvs. ethvert element i B vil også være medlem i A. I objektorientert sammenheng snakker man om superklasse (den generelle) og subklasse (den spesialiserte).

I vårt eksempel er ”Barnebok” en subklasse av ”Bok”, dette vises ved et trekantsymbol ved superklassen og en linje derfra til subklassen. Motivasjonen for å skille ut barnebøker i en spesiell subklasse, kan være at de har spesielle attributter eller metoder som ikke er aktuelle for bøker flest, som her *anbef_alder* og *selvles?* – den siste indikerer om barn i anbefalt alder kan antas å lese boken på egen hånd, eller om den må leses høyt av en voksen. Barnebøker vil også være plassert i en egen avdeling i biblioteket, og det kan være andre regler med hensyn på frister og antall man kan låne på en gang.

I figur 2 har superklassen bare en subklasse, men ofte vil det være flere alternative subklasser. Da vil man knytte alle disse til det samme trekantsymbolet. I mer avanserte tilfeller kan en og samme superklasse være generalisering for flere uavhengige subklassehierarkier på samme tid, da vil man også trenge flere trekantsymboler ved samme klasse. Figur 3 viser et eksempel hvor vi har flere ortogonale (dvs. uavhengige) dimensjoner som bøker kan spesialiseres etter, på den ene siden ”Barnebok” og ”Ungdomsbok”, på den andre siden ”Lånbar bok” og ”Ikke-lånbar bok” (f.eks. oppslagsverk som det ikke er lov å ta ut av biblioteket). I dette tilfellet annoterer man hver generaliseringstrekant med en viss *rolle*, her ”aldersgruppe” og

”lånbarhet”. Sistnevnte rolle er markert med {complete}, dette betyr at enhver bok må være enten lånbar eller ikke-lånbar. For ”aldersgruppe” fins ingen slik markering, altså vil man også kunne ha bøker som verken er barnebøker eller ungdomsbøker (f.eks. voksenbøker, som man ikke har sett det hensiktsmessig å lage noen egen klasse for).



Figur 3: To ortogonale generaliseringshierarkier

I noen tilfeller kan man ønske å gjøre superklassen *abstrakt*, hvilket innebærer at alle objekter som tilhører klassen også tilhører en eller annen av dens subklasser, slik at man ikke har til hensikt å instansiere objekter av selve superklassen. Dette kan markeres ved å sette nøkkelordet {abstract} høyrejustert under klassens navn, eventuelt også sette navnet i kursiv. Grensesnitt kan markeres nesten på samme måte som abstrakte klasser. Man bruker samme trekant som for generalisering under rektanglet for grensesnittet, og setter også her navnet i kursiv, men skriver nå nøkkelordet ”interface” venstrejustert over navnet. Dessuten er linjen fra trekanten til rektanglet for implementerende klasser stiplet. Av plasshensyn viser vi ikke eksempler på disse tingene. Abstrakte klasser og grensesnitt bør være forstått fra Programmeringsfaget, og spesielt interesserte kan eventuelt oppsøke mer inngående litteratur om UML for eksempler.

C) Aggregering. Denne relasjonen uttrykker at deler settes sammen til en større helhet, i invers forstand ofte kalt en ”part-of” relasjon. Hvis A er en aggregering av B og C, uttrykker dette i mengdealgebra at $A \subset B \times C$, dvs. at A er en undermengde av det kartesiske produktet av B og C. Det kartesiske produktet gir alle mulige kombinasjoner av de to operandene. Hvis f.eks. B er HTML-hoder (alle eksisterende tekststrenger ”<HEAD>…</HEAD>”) og C er HTML-kropper (alle eksisterende tekststrenger ”<BODY>…</BODY>”), vil det kartesiske produktet være alle mulige kombinasjoner hvor et hode settes sammen med en kropp. Mange av disse kombinasjonene vil ikke tilsvare eksisterende websider (selv om man i noen tilfeller kan tenke seg samme kropp eller samme hode duplisert flere steder). Nettopp derfor er det viktig å få med undermengdetegnet: Hvis A er klassen av websider vil det være riktig (hvis vi ser bort fra en del komplikasjoner, f.eks. at man også kan ha FRAMESET istf. BODY) å si at denne er et subsett av det kartesiske produktet.

I vår Figur 2 er aggregering brukt for å vise at et bokverk består av et antall bøker. For aggregering bruker UML et rombeformet symbol under helhetsklassen, med koblinger til delklassen(e). I dette tilfellet, hvor det bare er én delklasse og vi har kardinalitetsuttrykket 2..* på Boksiden av relasjonen, sier modellen at et bokverk består av minst 2, muligens flere, bøker (dvs. $Bokverk \subset Bok \times \ldots \times Bok$). På bokverksiden av relasjonen står det 0..*. Biblioteket kan godt ha bøker som ikke er med i noe bokverk (vanligvis de fleste). *-tegnet impliserer at en bok faktisk også kan være med i flere bokverk på en gang (f.eks. at Hamsuns ”Sult” kan være representert både i ”Hamsuns samlede” og ”Norske klassikere”).

Man kan godt tenke seg eksempler hvor flere delklasser er tilknyttet samme aggregering (en webside består av et hode og en kropp; en bil består av karosseri, chassis, motor, eksosanlegg, et antall hjul, etc.). I slike tilfeller kan man knytte flere delklasser til samme aggregeringsrombe, nettopp som man knyttet flere alternative subklasser til samme trekant for generalisering. Man kan også tenke seg muligheten av å dekomponere samme helhet på flere ortogonale måter. På den ene siden kan et menneske tenkes å bestå av hud, hår, skjelett, tenner, negler, muskler, blod, etc. (dekomponering i henhold til materialtype), på den annen side av hode, torso, armer, ben etc. (dekomponering i henhold til kroppsdel).

I UML fins det to forskjellige aggregeringsrelasjoner. Den ene kalles simpelthen *aggregation*, og vises ved hvitt rombesymbol, som brukt i vår figur. Den andre kalles *composition*, med samme rombesymbol, bare at det nå er fylt med svart. Betydningen er den samme, bortsett fra at composition-relasjonen har noen tilleggskrav:

- Ethvert delobjekt kan kun tilhøre én helhet. I vårt eksempel så vi en mulighet for at en bok kunne tilhøre flere bokverk på samme tid, dermed passet ikke ”composition” i dette tilfellet. Men svært ofte når det er snakk om aggregering, vil kravet om kun én helhet være implisitt. Et hjul kan f.eks. kun befinne seg på en bil av gangen.
- Delene er forventet å være avhengige av helheten for å eksistere. Dvs., hvis helheten forsvinner, så forsvinner også alle dens deler. I data(base)sammenheng vil dette innebære at man bruker kaskadesletting for delene dersom helheten blir slettet. Men kanskje er heller ikke dette ønskelig i vårt eksempel. Hvis man i utgangspunktet har et bokverk bestående av syv bøker, og så mister fem av dem, ville det kunne være aktuelt å slette bokverket (dvs. at datasystemet ikke lenger skal oppfatte det som om dette innehas av biblioteket). Normalt vil man likevel ikke ønske å kaste de enkeltbøkene man fremdeles er i besittelse av, men beholde disse som frittstående bøker.

Altså passer det ikke å bruke *composition* i vårt eksempel, og vi nøyer oss med vanlig *aggregation*. Men man kan lett tenke seg andre eksempler hvor det ville passe, f.eks. hvis en ordre består av et antall ordrelinjer (linje 1: 1000 eks av varenummer 24, linje 2: 500 eks av varenummer 56, etc.). Om man da sletter selve ordren, vil man også ønske å slette alle dens ordrelinjer, siden disse ikke kan eksistere uavhengig av noen ordre. Og i dette tilfellet ville en ordrelinje bare kunne være del av én ordre. Selv om det tilfeldigvis fantes to forskjellige kunder som begge hadde bestilt 1000 eks av varenummer 24, ville man likevel ikke si at dette var samme ordrelinje, men to forskjellige ordrelinjer med samme verdi for varenummer- og antall-attributtene, akkurat som det også kan finnes personer med samme navn, samme høyde etc., uten at disse dermed er den samme personen).

Det er viktig å være klar over forskjellen på generalisering og *klassifisering*. Generalisering er et forhold mellom to mengder (eller i objektorientert forstand: mellom to klasser), og forteller at den ene er en undermengde (subklasse) av den andre. Klassifisering er derimot et forhold mellom individ (enkeltobjekt) og klasse. Generalisering: *En barnebok er en bok*. Klassifisering: *”Sult” er en bok*. Det som kan lure enkelte, er at frasen ”er en” kan benyttes i begge sammenhenger. Men i det første tilfellet er det generelle substantiv (fellesnavn) både foran og bak, mens det i det andre tilfellet begynner med et særnavn (bokens tittel). Om man skulle klassifisere videre fra ”barnebok” heller enn å generalisere, ville man ikke komme til ”bok”, men få *barnebok er et substantiv* eller *barnebok er et begrep* – med klassifisering klarer man kun et par transitive ledd før man ender opp i ekstremt abstrakte konsepter.

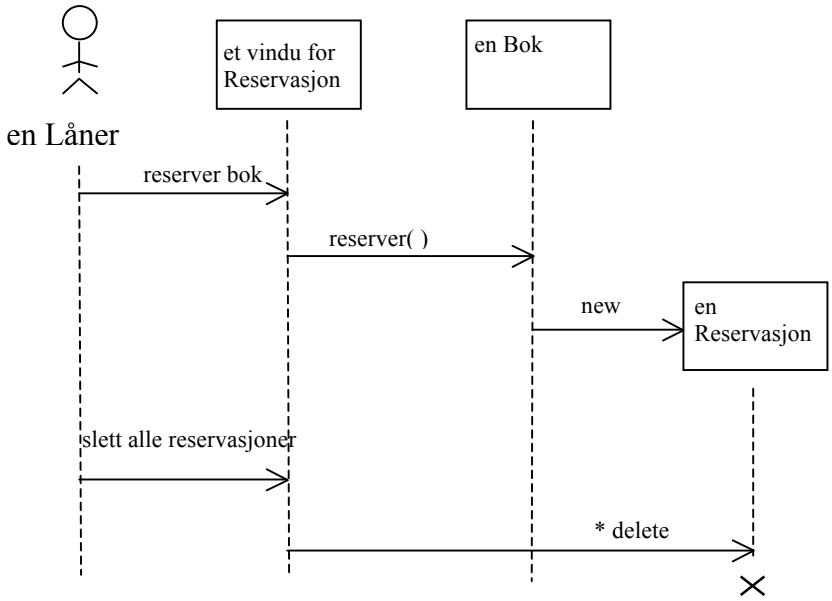
Det er også viktig å skjønne forskjellen på generalisering og aggregering, som noen har en lei tendens til å forveksle. Generalisering brukes som en relasjon mellom et generelt begrep og et

mer spesielt begrep. Et objekt av subklassen kan også sies å være medlem av superklassen, men ikke nødvendigvis omvendt. Aggregering brukes derimot som en relasjon mellom noe stort og noe mindre som dette store består av. I dette tilfellet vil ikke et objekt av en av delklassene også være et eksempel på et objekt av den overordnede klassen (et hjul er ikke en bil). For generalisering kan man som nevnt lage fornuftige setninger av typen ”En/et <subklasse> er en/et <superklasse>”, f.eks. ”En barnebok er en bok” (pass på å ha med artikler en/et, dette forhindrer forveksling med klassifisering). Prøver man derimot ”En bok er et bokverk”, eller ”Et bokverk er en bok”, vil forhåpentligvis de fleste føle motvilje. For aggregering kan man lage setninger som ”En/et <aggregatklasse> består av <delklasser> (hvor man bruker entall for delklassenavnet hvis kardinaliteten er 1, flertall hvis den er mange)”, f.eks. ”Et bokverk består av bøker”, ”En bil består av karosseri, chassis, motor, lykter, hjul…” Om man prøver med ”En bok består av barnebøker” eller ”En barnebok består av bøker”, vil man forhåpentligvis merke at noe ikke stemmer…

Et mer kinkig problem kan være å skjønne forskjellen på aggregering og vanlige assosiative relasjoner. Forholdet mellom Bokverk og Bok kunne også ha blitt modellert som en vanlig en-til-mange relasjon (eller i vårt tilfelle faktisk mange-til-mange, siden vi tillater bøker å være med i flere bokverk samtidig). I og for seg ville det ikke ha vært noe galt i dette, man kunne ende opp med en helt kurant design, siden kodingen godt kan bli den samme om man hadde aggregering eller vanlig assosiativ relasjon. Men konseptuelt sett blir modellen mindre informativ hvis man unnlater å bruke aggregering når det egentlig er dette man har. Med en vanlig relasjon mellom bokverk og bok ville vi ikke se noen fundamental forskjell på denne og øvrige relasjoner i diagrammet, f.eks. mellom låner og reservasjon. Men normalt ville man ikke oppfatte reservasjonene som en *del av* den respektive låneren, og om en reservasjon slettes, ville man ikke si at låneren som sådan hadde forandret seg. Hvis derimot en av bøkene i et bokverk går tapt for biblioteket, er bokverket som helhet blitt redusert (til et ”innkomplett bokverk”). På spørsmålet ”Har dere en låner som heter Kari Karinesdatter?” ville systemet eller bibliotekspersonellet uten videre kunne svare ja, såframt denne var registrert og den som spurte hadde rett til å få opplysningen – også om den nevnte Kari p.t. ikke hadde noen løpende reservasjoner eller lån. På spørsmålet ”Har dere Hamsuns samlede?” kunne det derimot innvendes at et ubetinget ”Ja” ville være galt svar, eller iallfall misvisende, hvis det faktisk var slik at noen av bindene var gått tapt. Da ville et korrekt svar snarere være ”Ja, men vi mangler bind 5 og 7.”– eller hvis mange av bindene manglet, kanskje til og med ”Nei, vi har bare de to første bindene.” En modell hvor man har et korrekt skille mellom vanlige assosiative relasjoner og aggregering/komposisjon, vil derfor gi bedre innsikt i forholdet mellom de involverte klassene.

8.4 Sekvensdiagrammer

Sekvensdiagrammer (*sequence diagrams*) har som mål å vise hvordan flere objekter samarbeider for å løse en bestemt oppgave. En ”oppgave” i denne sammenhengen vil typisk tilsvare ett brukstilfelle. I figur 5 er det vist to enkle sekvenser i det samme diagrammet. Her får vi illustrert både hvordan man viser metodekall mot eksisterende objekter, nyskaping av objekter og sletting av objekter. De sentrale konseptene i sekvensdiagrammet vil bli forklart i de følgende delkapitler.



Figur 5: To sekvenser i samme diagram

8.4.1 Objekter

Objekter vises ved navngitte rektangler som i utgangspunktet plasseres øverst i diagrammet. At det står ”en Bok”, ”en Reservasjon” etc., signaliserer nettopp at det her ikke dreier seg klasser, men om enkeltobjekter av klassene. Hvis man vil kan man også ha med aktører på linje med objektene for å vise hvordan handlingssekvensene initieres ved eksterne kommandoer fra brukeren. Aktørene er vist ved samme notasjon og har samme betydning som før, og vi vil ikke diskutere dem nærmere her, mange bruker dem heller ikke i sekvensdiagrammene. Den observante leser vil ha merket seg at vindusobjektet (det lengst til venstre) ikke er inkludert i klassediagrammet vårt i figur 3. Dette objektet er av en klasse som er del av brukergrensesnittet. Det er ikke noe i veien for å ha med slike objekter i klassediagrammene (tvert imot, de bør modelleres på et eller annet tidspunkt i utviklingen), men vi droppet for enkelthets skyld alle tanker på dette i figur 2 for å konsentrere oss om den konseptuelle siden ved biblioteksdomenet.

Hvert objekt vil være utstyrt med en *livslinje*, nemlig den stiplede linjen som går loddrett nedover fra hvert objekt. Disse indikerer objektets levetid, og står som et tilknytningspunkt for meldinger. I noen varianter av sekvensdiagrammer bruker man en mer detaljert notasjon hvor det er uthevede partier på livslinjen i de perioder det tilhørende objektet er aktivt, men dette er droppet her. Når det gjelder rekkefølgen på hendelser skal livslinjene leses ovenfra og ned. Et kryss på livslinjen betyr at objektet opphører å eksistere.

8.4.2 Meldinger

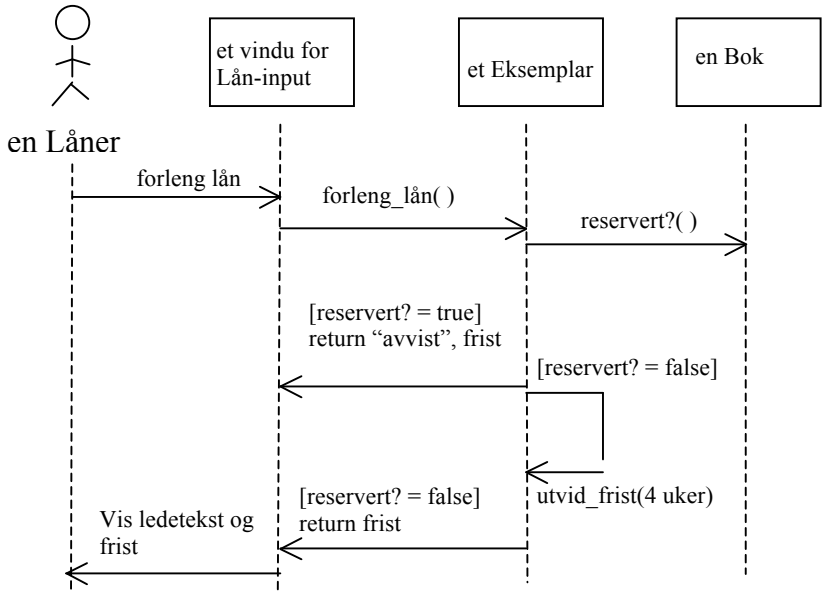
Meldinger vises ved piler mellom livslinjene, annotert med meldingens navn. En * foran meldingsnavnet betyr at meldingen kan bli repetert mange ganger. I noen tilfeller kan en melding gå rett til en objektboks i stedet for til dets livslinje, dette brukes når dette objektet blir opprettet (med *new*).

I figur 5 ser vi først at låneren gir en kommando om å reservere en bok (f.eks. ved å fylle inn opplysninger om forfatter og tittel og klikke på en ”Reserver”-knapp på skjermen). Dette fører til at vindu-objektet sender en *reserver()*-forespørsel til det aktuelle bok-objektet. Her ser vi for enkelhets skyld bort fra komplikasjoner som kunne føre til at reservasjonen ble avvist. Det

fortsetter dermed betingelsesløst med at bok-objektet utfører en *new*-operasjon som oppretter et reservasjonsobjekt.
I den neste sekvensen, som er satt sammen med den første så vi skal slippe å tegne så mange bokser, bestemmer en låner seg for å slette alle sine reservasjoner. Dette vil føre til en repetert (*) delete-operasjon mot diverse reservasjonsobjekter, og disse opphører å eksistere som følge av handlingen.

8.4.3 Betingelser

For mer presis modellering kan man både angi parametere og dessuten utstyre meldingene med *betingelser*, angitt i [ ]-parenteser. Hvis en betingelse er angitt, betyr dette at meldingen kun vil bli sendt dersom betingelsen er tilfredsstilt. Et eksempel med bruk av betingelser er vist i figur 6.



Figur 6: Sekvensdiagram med betingelser

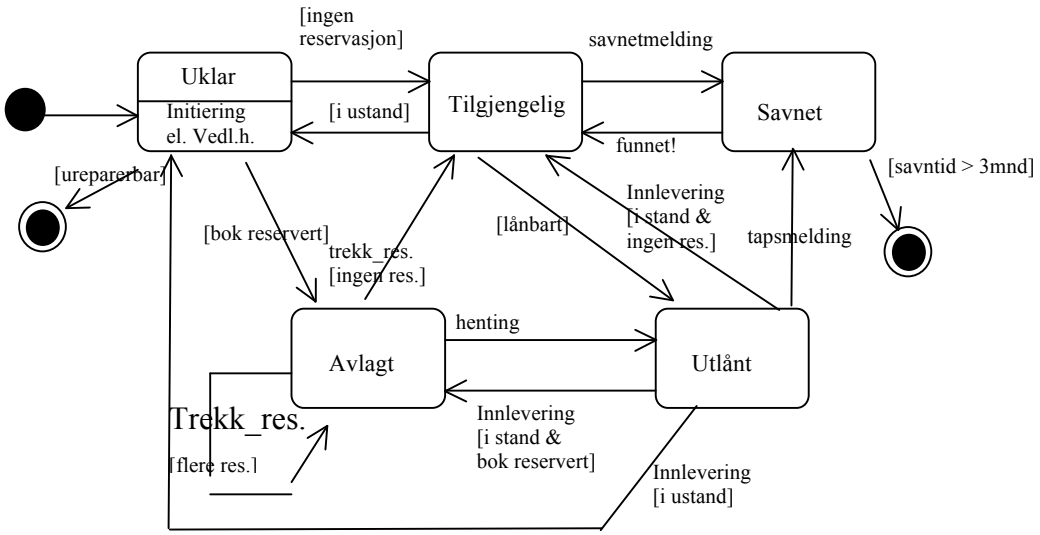
I figur 6 gir en låner inn et ønske om å forlenge lånet av et eksemplar, dvs. å få utsatt innleveringsfristen. La oss anta at bibliotekets policy på dette er at man standard får 4 ukers utsettelse, såfremt boken ikke er reservert av noen andre. Vinduet sender en *forleng_lån( )*-melding til det aktuelle eksemplar-objektet. Eksemplar-objektet må så sende en *reservert?( )*-melding til det tilhørende bok-objektet, denne vil mest hensiktsmessig returnere true eller false. Nå kommer bruken av betingelser inn i bildet. Hvis boken var reservert, må forlengelsen avvises. Ellers er det i orden å forlenge lånet. Eksemplar-objektet sender da meldingen *utvid_frist(4 uker)* til seg selv (dvs. kaller en metode i samme objekt), dette kalles selv-delegering (*self-delegation*). For de øvrige metodekallene har vi ikke inkludert parametere, enten fordi de ikke trengs, eller fordi de antas å være innlysende. Her har vi også valgt å vise hvordan resultater returneres, noe som ofte ikke gjøres i enkle sekvensdiagrammer (kanskje også fordi det ikke returneres noe, eller fordi det anses for innlysende hva som returneres). Hvis boken var reservert, gis melding tilbake i vinduet om at forlengelsen er avvist, samt at låneren påminnes om den gjeldende fristen for innlevering. Ellers gikk forlengelsen i orden, og det vises informasjon om den nye fristen.

Hvis det blir mye betingelser i sekvensdiagrammer, blir de fort uoversiktlige. De fleste eksperter anbefaler heller at man prøver å holde sekvensdiagrammene enkle og bruker

aktivitetsdiagrammer der det er behov for å vise flere alternative handlingsmåter. Aktivitetsdiagrammer blir diskutert på slutten av neste seksjon.

8.5 Tilstandsdiagrammer

Med tilstandsdiagrammer (*state diagrams*, evt. *state transition diagrams*, STD) er man på vei mot en mer presis, formell modellering av systemet, noe à la tilstandsautomater, som vil være kjent fra fag som Diskret matematikk. Mens sekvensdiagrammer egner seg for å vise meldingsutveksling og samarbeid mellom flere objekter, er et tilstandsdiagram best egnet til å vise oppførselen internt i et objekt, i form av alle mulige tilstander dette objektet kan ha, og hvilke overganger som er mulige mellom disse.
Figur 7 viser et tilstandsdiagram for eksemplar-objekter. Konseptene vil bli forklart i de påfølgende delkapitlene.



Figur 7: Tilstandsdiagram for eksemplar-objekt

8.5.1 Tilstander

Tilstander (eng. states) vises som firkanter med avrundede hjørner, og med navn som forklarer hva slags tilstand det er, dette vil typisk være adjektiver. Om man vil, kan man i stedet for selve tilstanden angi hvilken aktivitet som foregår mens objektet er i tilstanden, eller eventuelt angi både tilstand og aktivitet. Dette har vi gjort for tilstanden øverst til venstre, ”Uklar”. Et eksemplar vil befinne seg i denne tilstanden enten med det samme det er anskaffet, da man må initiere det (sette på strekkode og magnetstripe for tyverisikring), eller hvis slitasje gjør det nødvendig med vedlikehold. Således er aktiviteten her ”Initiering eller vedlikehold. Forøvrig har vi holdt oss til å kun navngi tilstander. ”Tilgjengelig”, ”Utlånt” og ”Savnet” burde være selvforklarende. ”Avlagt” innebærer at eksemplaret befinner seg i biblioteket, men er reservert av noen. Som vi husker fra klassediagrammet, gjaldt reservasjoner egentlig bøker, ikke eksemplarer, men den som har reservert, vil jo i så fall ønske det første eksemplaret biblioteket får kloa i, dette blir altså avlagt (f.eks. lagret ved disken, hvor låneren kan hente det).

8.5.2 Start og stopp

Start og *stopp* vises ved svarte rundinger, stopp med en hvit rand rund. Pilen fra Start-rundingen viser hvilken tilstand objektet vil befinne seg i når det oppstår, i dette tilfellet

”Uklart”. Piler som går til Stopp-rundinger viser fra hvilke tilstander objektet kan opphøre å eksistere, i dette tilfellet fra ”Uklar” og ”Savnet”.

8.5.3 Overganger

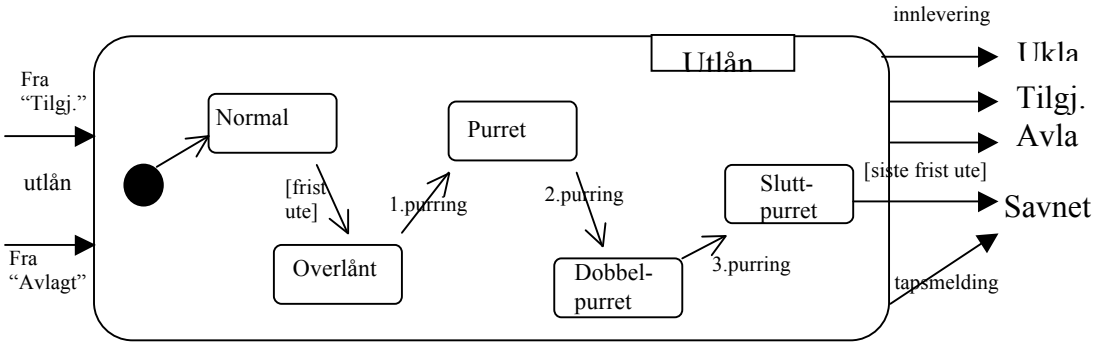
Overganger (eng. transitions) vises som piler mellom to tilstander, eller mellom en tilstand og start/stopp. Pilens retning forteller hvilken vei overgangen går, og den vil være annotert enten med navn på hendelsen som fører til overgangen og/eller en *betingelse* for når overgangen vil skje. Betingelser står i [ ]-parenteser, mens navn på hendelser står uten. I vårt eksempel er det altså noen overganger som bare har hendelsesnavn (f.eks. fra ”Tilgjengelig” til ”Savnet”) og noen som bare har betingelser (f.eks. fra ”Tilgjengelig” til ”Uklar”), og noen som har begge deler. Dette er valgfritt, man tar det som best illustrerer handlingsgangen.

Mens objektet tilbringer tid i de forskjellige tilstandene, er overgangene formelt sett ment å være momentane, dvs. ta null tid. Med hendelsen ”innlevering” mener man altså ikke hele handlingsgangen fra låneren kommer fram til disken og sier ”Jeg ønsker å levere inn denne boka” til innleveringen faktisk er blitt registrert og samtalen avsluttes med noen høflighetsfraser. Nei, her representerer hendelsen kun det øyeblikket akkurat da eksemplaret går fra å være utlånt til innlevert fra datasystemets synsvinkel, f.eks. ved at en statusverdi endres i databasen. Overgangen representerer altså kun det momentane skiftet av tilstand, ikke hva man gjør for å oppnå dette skiftet. Eventuelle betingelser innebærer derfor heller ikke at det gjøres noen vurderingen underveis på pilen (siden dette jo ville ta tid) – vurderinger av betingelser må eventuelt ha skjedd mens objektet fortsatt befant seg i fra-tilstanden.

Vårt eksempel kan nå forklares. Eksemplaret starter i den nevnte ”Uklar”-tilstanden. Når det er klargjort, vil det bli ”Tilgjengelig”, med mindre noen allerede har rukket å gjøre en reservasjon. I så fall blir eksemplaret ”Avlagt” (f.eks. lagret ved disken, i påvente av henting) for den som har reservert det. Når denne henter det avlagte eksemplaret, vil det bli ”Utlånt”, det samme skjer med et tilgjengelig eksemplar ved utlån. Her er betingelsen at eksemplaret er ”lånbart” – leksika og andre oppslagsverk vil det normalt være ulovlig å ta ut av biblioteket, men de er likevel tilgjengelige for studier på stedet. Når et utlånt eksemplar leveres tilbake, kan en av tre overganger inntreffe. Hvis eksemplaret er i ustand, sendes det på reparasjon, dvs. det blir ”Uklart” (dessuten må vi tro at låneren får et erstatningskrav, men dette vil ikke være med i vårt tilstandsdiagram, som bare tar for seg det som skjer med eksemplar-objektet). Hvis eksemplaret er i stand, og det ikke foreligger noen reservasjon av boken, vil det igjen bli tilgjengelig. Om det derimot fins en utilfredsstilt reservasjon for boken, vil det bli ”Avlagt”. Om den som har reservert, trekker reservasjonen tilbake, eller ikke henter eksemplaret innen en viss frist, blir det igjen ”Tilgjengelig”, med mindre det fins flere reservasjoner, da blir det ”Avlagt” til den neste i køen. Hvis en låner melder et eksemplar tapt, eller fortsatt ikke leverer etter et visst antall purringer, blir det ”Savnet”. Det samme kan skje med eksemplarer som ikke var utlånt, hvis man plutselig ikke finner dem på forventet plass biblioteket (kan være stjålet, feilplassert, feilaktig kastet, hvem vet?) Siden savnede eksemplarer kan dukke opp igjen, ønsker man ikke å slette objektet straks det meldes savnet eller tapt, i stedet venter man 3 måneder før man sletter objektet. Sletting kan også skje hvis man bestemmer seg for å kaste eksemplaret etter å ha funnet ut at det ikke lenger er verdt å reparere i tilstanden ”Uklar”.

Hvis tilstandsdiagrammene blir komplekse, kan det lønne seg å dekomponere dem, dvs. at noen tilstander kan ha sub-tilstander inni seg. I vårt første eksempel har vi overhodet ikke modellert komplikasjonene som inntreffer hvis en låner ikke returnerer et eksemplar i tide, annet enn at det til sist kan gå over til ”Savnet”. En mulighet kunne være å skille mellom to tilstander, en kalt ”Utlånt” som før, og en kalt ”Overlånt”, som innebærer at låneren har

oversittet fristen. Hvis biblioteket har et fast system for purringer, f.eks. at man først venter til det har gått en uke over fristen, deretter sender ut første purring, så venter i ytterligere to uker før man sender ut andre purring, og deretter venter ytterligere tre uker før man sender siste purring, med beskjed om at låner må betale erstatning hvis man ikke leverer innen en uke. For en presis modellering kunne man her operere med en tilstand for hvert stadium i denne purringsprosessen. ”Overlånt” kan bety at fristen er utgått (men fremdeles med mindre enn en uke, slik at man ikke har purret ennå, hvilket innebærer at låneren kun får forsinkelsesgebyr, ikke purregebyr), deretter har man tilstandene ”Førstegangspurret”, ”Andregangspurret” og ”Sluttpurret”. Selvsagt kunne vi inkludere alle disse tilstandene på samme nivå som de vi allerede har i figur 7, men dette ville bli nokså uoversiktlig. Et bedre alternativ er vist i figur 8, hvor vi som før har en ”Utlånt”-tilstand, bare at denne er dekomponert i fem sub-tilstander (”Normal”, som betyr at man fortsatt er innen fristen, pluss de fire stadiene av overskridelse).



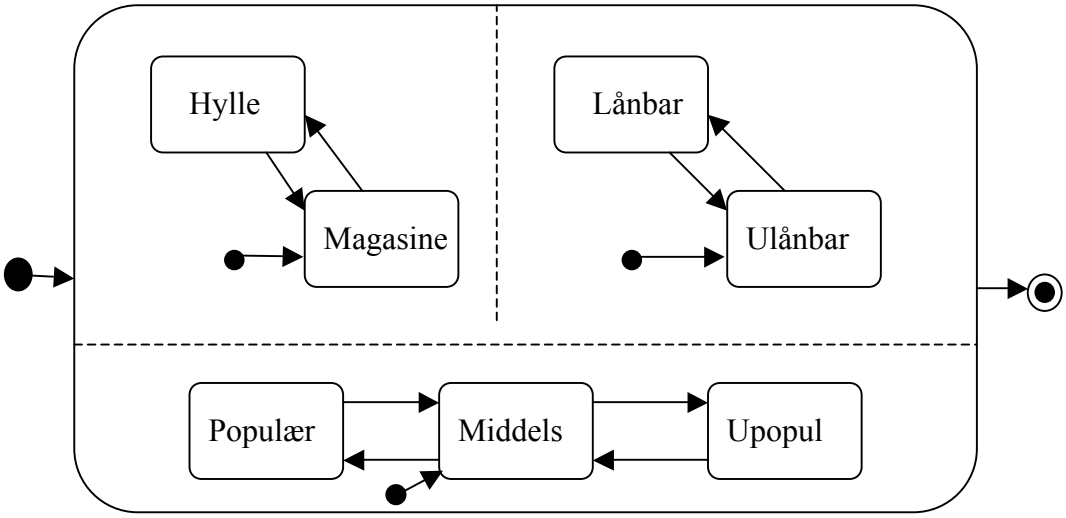
Figur 8: En dekomponert tilstand

Som vi ser går pilene på venstre side inn til supertilstanden ”Utlånt”, som er navngitt i et lite rektangel i kanten av den store boksen, og ikke til noen av subtilstandene. Da trenger vi et nytt start-symbol inni ”Utlånt”-tilstanden for å vise at man begynner i tilstanden ”Normal” (dvs. vanlig utlån, innen fristen). Alternativt kunne vi her ha kjørt pilene utenfra rett til ”Normal”, dette er en smakssak. Her har vi uansett ikke tjent noe i forhold til å modellere uten dekomponering. Fordelen med dekomponering kan derimot ses på høyre side. De tre pilene for ”innlevering” og den ene for ”tapsmelding” går her fra supertilstanden ”Utlånt” og ikke fra subtilstandene. Låneren kan jo på et hvilket som helst stadium i purreprosessen komme til å levere tilbake eksemplaret, eller melde det tapt. At pilene går fra supertilstanden indikerer nettopp dette, at overgangene kan skje uansett hvor man befinner seg inni den dekomponerte tilstanden. Hvis vi skulle ha modellert alle disse tilstandene uten bruk av dekomponering, måtte vi hatt tre innleveringspiler og en tapspil fra hver eneste av disse tilstandene. Dette ville gi et temmelig kaotisk diagram. Nå er det kun pilen til ”Savnet” i tilfelle låneren lar fristen gå ut uten å gi respons, som må gå fra en intern tilstand. Dette gjør ikke noe, for denne overgangen er kun aktuell fra ”Sluttpurret”, så den resulterer bare i en pil likevel.

Når man dekomponerer en tilstand i flere sub-tilstander, er det fortsatt meningen at objektet bare kan befinne seg i én av disse tilstandene om gangen. Men i noen tilfeller kan det også være interessant å operere med *parallelle* tilstander, hvor altså objektet kan være i flere tilstander på samme tid. Dette brukes når objektet kan gjennomgå flere forskjellige typer tilstandsendringer som er *uavhengige* av hverandre. Anta for eksempel at en bok kan være "Hyllet" (plassert i bibliotekets hyller, slik at lånere kan finne den selv) eller "Magasinert" (plassert i magasin, slik at det trengs en spesiell forespørsel til personellet for å få tak i den). En bok kan også være "Lånbar" eller "Ulånbar" (det siste vil gjerne være tilfelle for leksika og

lignende, evt. også bøker som pga. alder eller sjeldenhet har nådd en betydelig verdi). For det tredje kan en bok være registrert som "Populær", "Middels" eller "Upopulær", alt etter hvor stor etterspørsel det er etter den (for eksempel basert på oppsamlede data, eller bibliotekspersonellets subjektive oppfatning). La oss her se bort ifra de eksakte kriteriene for at det skal skje endringer fra den ene tilstanden til den andre, men anta at de tre nevnte dimensjonene er uavhengige av hverandre. Dvs., en bok kan være "Hyllet, lånbar og populær", "Hyllet, lånbar og upopulær", "Magasinert, ulånbar og upopulær", etc., etc. Uten bruk av parallelle tilstander ville vi nettopp være nødt til å operere med slike komplekse tilstandsbetegnelser, og i alt ville vi få 12 tilstander (2 x 2 x 3).

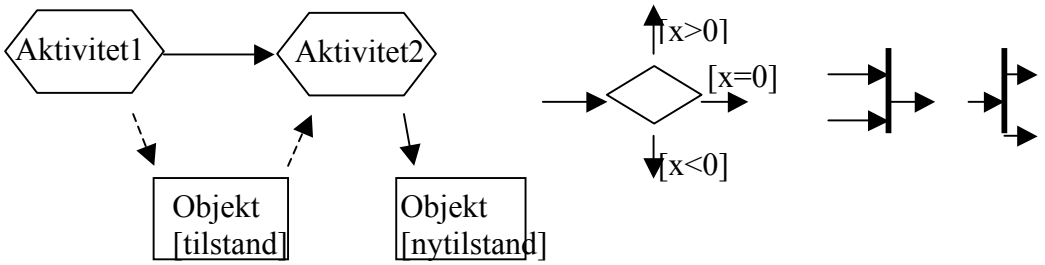
For å unngå en kombinatorisk eksplosjon av tilstander når man har uavhengige alternativ, er det bedre å bruke parallelle tilstander. I UML er dette gjort ved å dele opp en tilstandsboks med stiplede linjer, hvert avdelt område tilsvarer da en parallell. Dette er vist i Figur 9.



Figur 9: Tre parallelle tilstander

Aktivitetsdiagrammer kan oppfattes som en variant/utvidelse av tilstandsdiagrammer. Som nevnt i innledningen om tilstandsdiagrammer kunne vi inni en tilstandsboks navngi selve tilstanden og/eller den aktiviteten som foregikk i tilstanden. Aktivitetsdiagrammer vil altså være tilstandsdiagrammer der man kun, eller i hovedsak, opererer med aktiviteter. I den grad man fortsatt ønsker å inkludere selve tilstandsendringene i diagrammet, vil disse nå være vist som rektangler (med objektnavn, [tilstand]), koblet med stiplede piler til de aktiviteter som forårsaker tilstandsendringene. I tillegg har man en del andre konstruksjoner, f.eks. for å vise valgmuligheter, synkronisering og parallellprosessering.

For valg har man rombesymboler à la hva man vil være kjent med fra algoritmiske flytdiagrammer. I flytdiagrammer har man ofte bare to veier ut fra romben (f.eks. J og N, alt etter om betingelsen er sann eller usann), men i UMLs aktivitetsdiagrammer kan man ha piler ut, hver annotert med betingelsen for at denne skal velges, f.eks. [x<0], [x=0], [x>0]. Synkronisering og splitting vises begge ved en tykk stolpe som piler går inn til og ut fra. Flere piler inn og en ut betyr synkronisering – flere parallelle aktivitetstråder fortsetter i en felles. En pil inn og flere ut betyr splitting – en aktivitetstråd videreføres i flere parallelle aktiviteter. Denne symbolikken er vist i Figur 10.



Figur 10: Notasjon for aktivitetsdiagrammer

Man trenger ikke ha med objekt/tilstandsbokser for hver aktivitet, bare hvis man finner dette nødvendig for diagrammets forståelighet. Hvis man bare bruker aktiviteter, vil diagrammet minne mye om tilstandsdiagrammer, piler indikerer overganger på samme måte, og man har samme symbolikk for start og stopp. For eksempler på bruk av aktivitetsdiagrammer, se nedennevnte støttelitteratur eller forelesning om tilstandsdiagrammer.

8.6 Referanser

Dette korte notatet har bare kunnet gi en rask innføring i UML, med det som er mest nødvendig for å komme i gang med prosjektet. For alle diagramtypene som er nevnt her, fins det mer avanserte muligheter som kunne ha vært nevnt i tillegg, ikke minst det formelle språket OCL, som kan brukes til tekstlige presiseringer i forhold til diagrammene, med formler, variabeltilordninger, betingelser o.l. Det fins dessuten en del diagramtyper som ikke er vist, f.eks. aktivitetsdiagrammer (*activity diagrams*), samarbeidsdiagrammer (*collaboration diagrams*), pakkediagrammer (*package diagrams*) og realiseringsdiagrammer (*deployment diagrams*). Pakkediagrammer, som viser hvordan store systemer er delt opp i moduler, vil man se et eksempel på i kravspesifikasjonen for prosjektet. Det er ikke gitt at man finner *alt* man trenger å vite om UML-modellering i prosjektsammenheng i dette heftet, for mer detaljer henviser vi til forelesninger og den nedennevnte støttelitteraturen.

[1] Martin Fowler, Kendall Scott: *UML Distilled: applying the standard object-oriented modelling language,* 1997, 179 s. (ordet *distilled* = destillert hentyder på en rask innføring, så kan man bare tenke seg hvor destillert dette lille notatet da må være).
[2] Rob Pooley, Perdita Stevens: *Using UML: software engineering with objects and components*, 1999, 254 s.
[3] Bernd Oesterreich: *Developing Software with UML: object-oriented analysis and design in practice*, 1999, 321 s.
[4] Charles Richter: *Designing Flexible Object-Oriented Systems with UML*, 1999, 404 s.
[5] Ivar Jacobson, Grady Booch, James Rumbaugh: *The Unified Software Development Process*, 1999, 463 s. (en bok av metodens/språkets opphavsmenn)
[6] Desmond F. D'Souza, Alan C. Wills: *Objects, Components, and Frameworks with UML: the Catalysis Approach*, 1999, 785 s.

9 Vedlegg B - Use Case Points

Use Case Points (UCP) is an estimation method that provides the ability to estimate an application's size and effort from its use cases. Based on work by Gustav Karner in 1993, UCP analyzes the use case actors, scenarios and various technical and environmental factors and abstracts them into an equation.

The equation is composed of four variables:

1. Technical Complexity Factor (TCF).
2. Environment Complexity Factor (ECF).
3. Unadjusted Use Case Points (UUCP).
4. Productivity Factor (PF).

Each variable is defined and computed separately, using perceived values and various constants. The complete equation is:

```
UCP = TCF * ECF * UUCP * PF
```

The necessary steps to generate the estimate based on the UCP method are:

1. Determine and compute the Technical Factors.
2. Determine and compute the Environmental Factors.
3. Compute the Unadjusted Use Case Points.
4. Determine the Productivity Factor.
5. Compute the product of the variables.

9.1 Technical Complexity Factors

Thirteen standard technical factors exist to estimate the impact on productivity that various technical issues have on an application. Each factor is weighted according to its relative impact. A weight of 0 indicates the factor is irrelevant and the value 5 means that the factor has a strong impact.

Technical Factor	**Description**	**Weight**
T1	Distributed system	2
T2	Performance	1
T3	End User Efficiency	1
T4	Complex internal Processing	1
T5	Reusability	1
T6	Easy to install	0.5
T7	Easy to use	0.5
T8	Portable	2
T9	Easy to change	1
T10	Concurrent	1
T11	Special security	1

	features	
T12	Provides direct access for third parties	1
T13	Special user training facilities are required	1

Figure 1: Technical Factors.

For each project, the technical factors are evaluated by the development team and assigned a value from 0 to 5 according to their perceived complexity – multithreaded apps. require more skill and time than single threaded applications, for example, as do reusable apps. A perceived complexity of 0 means the technical factor is irrelevant for this project; 3 is average; 5 means that it has a strong influence.

Each factor's weight is multiplied by its perceived complexity to produce its calculated factor. The calculated factors are summed to produce the Total Technical Factor. To produce the final TCF, two constants are computed with the Total Technical Factor. The complete formula to compute the TCF is as follows:

```
TCF = 0.6 + (0.01*Total Technical Factor)
```

9.2 Environmental Complexity Factors

Environmental Complexity estimates the impact on productivity that various environmental factors have on an application. Each environmental factor is evaluated and weighted according to its perceived impact and assigned a value between 0 and 5. A rating of 0 means the environmental factor is irrelevant for this project; 3 is average; 5 means it has strong influence. Thus, if the participants are very familiar with UML, this will give value of 5 * weight = 7.5. In the same way, if the programming language is very well known, we get a value of 0 * weight = 0.

Environmental Factor	**Description**	**Weight**
E1	Familiarity with UML	1.5
E2	Application Experience	0.5
E3	Object Oriented Experience	1
E4	Lead analyst capability	0.5
E5	Motivation	1
E6	Stable Requirements	2
E7	Part-time workers	-1
E8	Difficult Programming language	-1

Figure 2: Example Environmental Factors.

Each factor's weight is multiplied by its perceived complexity to produce its calculated factor. The calculated factors are summed to produce the Total Environmental Factor. To produce the final ECF, two constants are computed with the Total Environmental Factor. The complete formula to compute the ECF is as follows:

```
ECF = 1.4 + (-0.03*Total Environmental Factor)
```

9.3 Unadjusted Use Case Points (UUCP)

Unadjusted Use Case Points are computed based on two computations:

1. The *Unadjusted Use Case Weight* (UUCW) based on the total number of activities (or steps) contained in all the use case Scenarios.
2. The *Unadjusted Actor Weight* (UAW) based on the combined complexity of all the use cases Actors.

UUCW

Individual use cases are categorized as Simple, Average or Complex, and weighted depending on the number of steps they contain - including alternative flows.

Use Case Type	Description	Weight
Simple	A simple user interface and touches only a single database entity; its success scenario has 3 steps or less; its implementation involves less than 5 classes.	5
Average	More interface design and touches 2 or more database entities; between 4 to 7 steps; its implementation involves between 5 to 10 classes.	10
Complex	Involves a complex user interface or processing and touches 3 or more database entities; over seven steps; its implementation involves more than 10 classes.	15

Figure 3: Use Case Categories.

The UUCW is computed by counting the number of use cases in each category, multiplying each category of use case with its weight and adding the products.

UAW

In a similar manner, the Actors are classified as Simple, Average or Complex based on their interactions.

Actor Type	Description	Weight
Simple	The Actor represents another system with a defined API.	1
Average	The Actor represents another system interacting through a protocol, like TCP/IP.	2
Complex	The Actor is a person interacting via an interface.	3

Figure 4: Actor Classifications.

The UAW is calculated by counting the number of actors in each category, multiplying each total by its specified weighting factor, and then adding the products.

Finally, the UUCP is computed by adding the UUCW and the UAW.

9.4 Productivity Factor

The Productivity Factor (PF) is a ratio of the number of man hours per use case point based on past projects. If no historical data has been collected, a figure between 15 and 30 is suggested by industry experts. A typical value is 20.

9.5 Final Calculation

The Use Case Points is determined by multiplying all the variables:

```
UCP = TCF * ECF * UUCP * PF
```

10 Vedlegg C - Brukbarhetstesting