Rapporter fra Høgskolen i Buskerud nr. ÌÌ

RAPPORT

RAPPORT

Ú^¦•ã c^}• Áæç Á[àb^\ c^¦

Ø[¦•\ b^||ã ^ Á{ èc^¦ Áè Áæ ¦^ Á[àb^\ c^¦ Á] è

T ^ åÁ^\ •^{] |^¦ ÁãÁÔ À

S} ˇ cÁY ÈÁP æ} • • [}

Rapporter fra Høgskolen i Buskerud

Nr. 88

Persistens av objekter

Forskjellige måter å lagre objekter på
Med eksempler i C#

Av

Knut W. Hansson

Hønefoss 2012

HiBus publikasjoner kan kopieres fritt og videreformidles til andre interesserte uten avgift.

En forutsetning er at navn på utgiver og forfatter(e) angis- og angis korrekt. Det må ikke foretas endringer i verket.

ISBN 978-82-8261-013-1 (trykt)
ISBN 978-82-8261-014-8 (online)

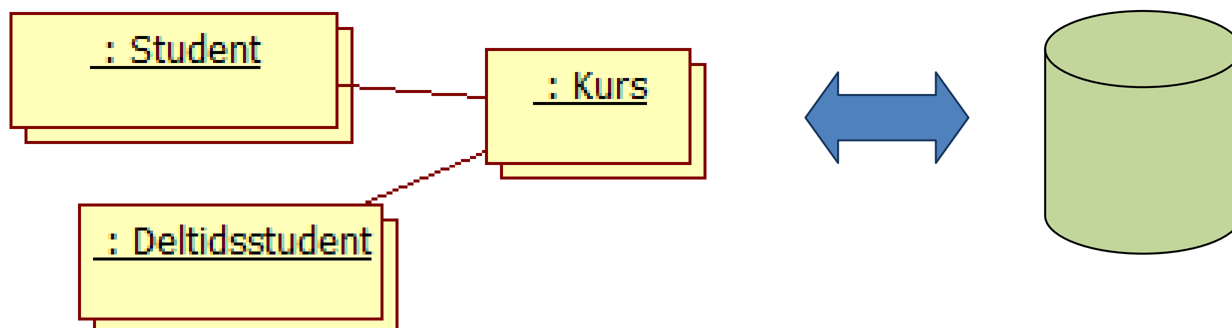
ISSN 0807-4488 (trykt)
ISSN 1893-2312 (online)

Persistens av objekter

Forskjellige måter å lagre objekter på
Med eksempler i C#

Knut W. Hansson
Førstelektor IT

6 February 2012



HiBus publikasjoner kan kopieres fritt og videreformidles til andre interesserte uten avgift.

En forutsetning er at navn på utgiver og forfatter angis – og angis korrekt. Det må ikke foretas endringer i verket.

Emneord:
objektorientert
lagring
database
eksempler
C#, C sharp

English keywords:
object oriented
storing
database
examples
C#, C sharp

Innhold

SAMMENDRAG/SYNOPSIS 1

Sammendrag 1

Synopsis in English 1

BAKGRUNN 3

Datasystemer sett på som en modell av virkeligheten 3

Objektorienterte datasystemer 3

Avbildning av objekters tilstand 4

Eksemplet som brukes 5

A: APPLIKASJONEN UTEN PERSISTENS 11

Programeksempel – uten persistens 11

B: LAGRING I FLAT, KOMMALIMITERT (CSV) FIL 13

Programeksempel – lagring i flat, kommalimitert (csv) fil 13

C: LAGRING AV SERIALISERTE OBJEKTER (BINÆR FORM) 17

Programeksempel – lagring i binær form 18

D: LAGRING I XML-FIL 21

D1: LAGRING I XML-FIL MED XMLSERIALIZER 23

Programeksempel – lagring i XML-fil med XmlSerializer 24

D2: LAGRING I XML-FIL MED DATACONTRACTSERIALIZER 27

Programeksempel – lagring i XMLfil med DataContractSerializer 27

E: LAGRING I EN RELASJONSDATABASE 29

Mapping 30

Normalisering av datamodellen 32

Skape databasen 32

Programeksempel E1: Lagring i relasjonsdatabase – programmerer alt selv 32

Programeksempel E2: Lagring i relasjonsdatabase med et hjelpeobjekt 36

Programeksempel E3: Lagring i relasjonsdatabase med Entity Framework o.a. verktøy 37

F: LAGRING I OBJEKTDATABASE (DB4O) 39

Prinsippene for db4os virkemåte 40

Programeksempel F1: Lagring med db4objects 42

Programeksempel F2: Søking med db4objects 44

Oppsummering om søking med Db40 49

G: PERSISTENS VED HJELP AV LOGGING (PREVAYLER) 51

Generelt 51

Tankegangen bak 51

Prevayler 52

Programeksempel: Persistens ved hjelp av logging (Prevayler) 52

OPPSUMMERING OM PERSISTENS AV OBJEKTER 55

Konklusjon for flat, kommalimitert (csv) fil 56

Konklusjon for serialiserte objekter (binær form) 56

Konklusjon for XML-fil 57

Konklusjon for relasjonsdatabase 57

Konklusjon for objektdatabase 58

Konklusjon for logging 59

Sammenlikning av persistensmetodene 59

VEDLEGG A – PROGRAMEKSEMPLET UTEN PERSISTENS 61

Sammendrag/synopsis

Sammendrag

Objektorienterte systemer skaper objekter i maskinens internminne. Når slike systemer avsluttes, vil minnet bli tømt og objektene går tapt. Man vil vanligvis sikre at objektene tas vare på til neste gang systemet åpnes – såkalt *persistens* av objektene. Det innebærer i praksis lagring til et ytre lager, gjerne et platelager.

Slik persistens kan gjøres på forskjellige måter. I denne rapporten omtales noen av disse måtene og det vises enkle eksempler på hvordan det rent faktisk kan gjøres i programspråket C# ("C-sharp").

De færreste persistensmetodene er objektorienterte. Eksemplene viser at metodene da har betydelige ulemper, fordi objektene må konverteres fra objektorientert til "flat" struktur. Det er tungvint for programmereren og kan fort føre til feil. Noen av metodene stiller dessuten urimelige krav til det objektorienterte systemet.

Det vises også to eksempler på objektorienterte persistensmetoder, herunder en objektorientert database, og da viser det seg at persistensen er ganske enkel for programmereren av det objektorienterte systemet. Det blir ingen konvertering og derfor også mindre sannsynlighet for feil.

Synopsis in English

Object-oriented systems create objects in the machine's internal memory. When such systems are closed, the memory will be cleared and the objects are lost. One will usually wish that the objects are saved until the next time the system is opened – so-called *persistence* of objects. In practice, this means storing the objects to an external storage, usually a hard disk.

Such persistence can be done in different ways. In this report some of these ways are discussed and some simple examples of how it actually can be done are shown, in the program language C # ("C sharp").

Few persistence methods are object oriented. The examples show that these methods have significant disadvantages, because the objects must be converted from object oriented to "flat" structure. This is cumbersome for the programmer and can easily lead to mistakes. Some of the methods also place unreasonable demands on the object-oriented system.

Two examples of object-oriented persistence methods, including an object-oriented database, are shown. It turns out that the persistence is quite easy for the programmer of the object-oriented system. Also, there is no conversion and therefore less likelihood of errors.

Bakgrunn

Datasystemer sett på som en modell av virkeligheten

Et administrativt, IT-basert informasjonssystem kan ofte ses på som en modell av virkeligheten omkring oss. Virkeligheten består av konkrete og abstrakte ting, hendelser, aktiviteter og forhold som kan kalles "objekter"/"entiteter"). Mellom disse er det sammenhenger, kalt "assosiasjoner"/"relasjoner". En utvalgt del av denne virkeligheten er av spesiell interesse for virksomheten kalles gjerne "Universe of Discours" (UoD). Objektene har mange "egenskaper"/"Attributter", og noen av disse velges ut som spesielt interesse. Likeartede objekter grupperes til "klasser"/"entitetstyper" med samme egenskaper.

Modellen er slik at hvert objekt i UoD representeres inne i systemet – det kan være som en post i en fil, som en rad i en databasetabell, som et objekt i et objektorientert system eller annet. (Eventuelt kan objektet deles på flere poster/rader/objekter.) Relasjonene modelleres ved at det settes opp sammenhenger mellom objektene i modellen i form av postpekere, fremmednøkler, fysiske pekere eller liknende. Alt ender på en eller annen måte opp som "data" i modellen.

Tanken er da at når virkeligheten endres, så endres modellen tilsvarende, slik at modellen hele tiden er et ajourholdt bilde av virkeligheten. Hvis vi vil vite noe om virkeligheten, kan vi da "avlese" modellen. Modellen gjør det også forholdsvis enkelt å gruppere store mengder, finne statistiske mål osv.

Et vanlig akronym innenfor databasefaget er CRUD (Create, Retreive, Update, Delete poster). Det betegner de operasjonene som er nødvendige for å holde modellen à jour med endringer i virkeligheten altså UoD. Uansett hvordan data lagres, er det avgjørende at disse operasjonene kan gjennomføres enkelt, effektivt og sikkert.

Av disse er det vanlig å anta at i administrative systemer er henting (retreive) vanligste[1]. I tillegg til at henting ofte er mer vanlig enn oppdatering, vil *alle* operasjonene (i CRUD) kreve at data hentes. Innsetting vil f.eks. kreve kontroll av at dataene ikke allerede finnes i modellen, oppdatering kan bare skje etter at det eksisterende er funnet, og det samme gjelder sletting. Det er følgelig viktig at henting kan gjøres enkelt og raskt.

Når mange klienter kjører mot de same datene på en tjener – som i relasjonsdatabaser – er det vanlig å kreve at alle oppdateringer skal være "ACID" (Atomic, Consistent, Isolated, Durable). Transaksjoner som flettes, men opprettholder ACID, vil ikke ødelegge for hverandre. Jeg går ikke nærmere inn på dette her, da jeg antar at det er kjent fra databaseteori.

Objektorienterte datasystemer

I objektorienterte systemer skapes, gjenfinnes, endres og slettes (ref. databasebegrepet CRUD) objekter. Mange av disse er entitetsobjekter, dvs. objekter som modellerer objekter av interesse for virksomheten. Særlig vil det være mange slike objekter i "administrative applikasjoner" (i motsetning da til systemer som styrer prosesser o.l.). Entitetsobjektene utgjør da sentrale elementer i modellen av UoD.

Når kjøringen av applikasjonen avsluttes, vil alle objektene i internminnet bli borte. Imidlertid ønsker man jo at entitetsobjektene blir bevart til neste gang applikasjonen startes – altså mellom sesjoner. *Dette kalles persistens*.

[1] Unntak er bl.a. typiske logger, der data neste bare legges til – gjerne sist i filen.

For å få persistens må objektene lagres på et ytre lager, f.eks. en harddisk. Det er flere måter å gjøre det på, så man må gjøre noen valg:

1. Alle tilstandsendringer krever en *melding* (*public* variable bør ikke forekomme, da de kan endres direkte uten melding). Derfor vil gjentakelse av alle sendte meldinger føre systemet tilbake til samme tilstand som da systemet stanset. Man kan følgelig velge om man vil lagre objektenes tilstand (verdien av variablene) eller lagre alle meldinger som går i systemet.
2. Man kan lagre alt når applikasjonen avslutter, eller lagre objekter hver gang de endres. Det første kan føre til at endringer går tapt hvis applikasjonen stopper ukontrollert, men i noen sammenhenger er ikke det så alvorlig. Det andre alternativet er tryggere, men gir også et tregere system da alle operasjoner mot ytre lager er forholdsvis trege.
3. Man kan lagre objektenes attributtverdier som data i flate filer med forskjellige strukturer, eller i en relasjonsdatabase. Når programmet starter igjen kan da dataene leses og objektene bygges opp igjen. Mer avansert vil det være å lagre *objektene*, enten i fil(er) eller i en objektorientert database. Da slipper man også å programmere gjenoppbyggingen av objektene.
4. Man må også velge når objektene skal hentes: Skal alle hentes ved programstart (*eager loading*) eller etter hvert som det er behov for dem (*lazy loading*).
5. Objektene refererer til hverandre i et nettverk (en *graf*). Det reiser spørsmålet om hvor langt ned i grafen man lagrer og henter. Hvis f.eks. et personobjekt har en streng *navn* som attributt, vil det være rimelig å lagre/hente strengattributtet sammen med personobjektet. Men anta nå at personene som modelleres eier biler som også er med i modellen. Da vil personobjektene referere til bilobjekter som personen eier. Bør man da lagre/hente alle disse bilobjektene når personobjektet lagres eller vente til personobjektet faktisk har behov for bilobjektene? Valget vil påvirkes av om objektene som det refereres til er entitetsobjekter i seg selv, eller bare må anses som egenskaper ved et annet objekt.
6. Vil man benytte klassebiblioteker eller programmere selv? Biblioteker har gjerne klasser som er gjennomtestet, raske og med få feil, men de er generelle og kan gi "overkill". Egne programmer, derimot, kan ha feil, men gir full kontroll. Dermed kan de bli bedre tilpasset og raskere.

I denne rapporten viser jeg og diskuterer flere måter å få til persistens på.

Avbildning av objekters tilstand

Med et objekts tilstand forstår vi verdien av alle objektets egenskaper (variabelverdiene) når objektet er "i ro"[2]. Grunnen til at objektet må være "i ro" når det avbildes, er at objektets egenskaper kan ha sammenheng seg imellom. Av hensyn til konsistens må objektet da gjøre seg ferdig med en tilstandsendring før avbildning tillates. Dette er analogt med kravet om at endringer skal være ACID (se ovenfor).

Vi vil altså ha en avbildning av objektet, der alle egenskapene er avbildet slik de var i ett og samme øyeblikk og var konsistente seg imellom.

Tilsvarende må vi sørge for at alle objektene som utgjør et system, avbildes utfra verdiene på samme tidspunkt. Det er fordi objektene refererer til hverandre og referansene endres. I eksempelet

[2] Noen vil også ta med objektets programteller i tilstandsbegrepet mens objektet gjør noe. Når objektet tilstand skal avbildes er det uansett avgjørende at objektet "er i ro", altså at det ikke er under endring. Objektet er da i "ventetilstand" og avventer neste melding (her inkluderer jeg da endring av synlige egenskaper fra utsiden). Da har objektet ingen programteller, så vi kan se bort fra den her.

mitt nedenfor, har studenter referanser til kurs de tar, samtidig som kursene refererer til studenter som tar kurset. Hvis avbildningen da f.eks. ender med at en student refererer til et kurs han/hun tar, uten at kurset refererer til studenten, så er avbildningen av *systemet* blitt inkonsistent. Igjen er dette analogt med ACID transaksjoner.

I en enbrukerapplikasjon uten tråder, vil alle handlinger – også avbildningen – skje med synkrone meldinger. Da kan vi enkelt sørge for at situasjoner som nevnt ovenfor ikke skjer. I praksis vil imidlertid applikasjoner ofte ha tråder, som i utgangspunktet eksekverer tidsdelt "semiparallelt". For det andre er systemene ofte delt i mange klienter og en/flere tjenere, der hver tjener betjener mange klienter. I begge tilfeller kan systemdeler forstyrre hverandre, så noen vil endre objektene samtidig[3] som andre vil avbilde dem.

Vi ønsker følgelig en form for låsing av objektene, evt. også hele eller deler av systemet mens det avbildes. I C# kan man låse objekter eksplisitt med setningen *lock(x)* der *x* er et objekt (en *struct* er da ikke tillatt):

```
lock (x)
{
    DoSomething();
}
```

Objektet må da være deklarert *private*.

Hvis man vil låse hele eller større deler av systemet, må man følgelig sørge for at objekter som skal avbildes er referert i ett, enkelt objekt, som så låses. Det finnes også andre måter å gjøre det på som jeg ikke tar med her.

I Java benyttes tilsvarende det reserverte ordet *synchronized*, f.eks.

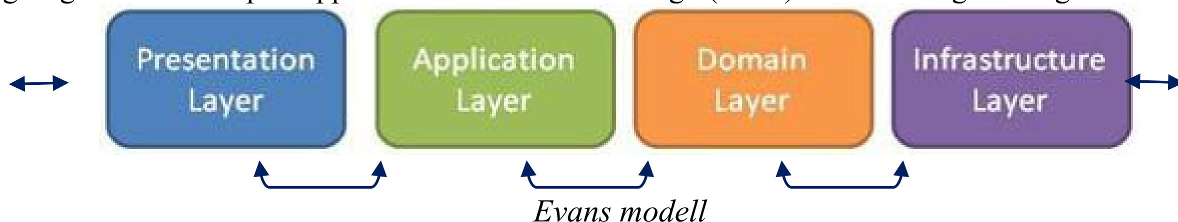
```
public synchronized void eksempel() {
   // beskyttet kode her
}
```

Eksemplet som brukes

Prinsipper

Som eksempel skal jeg lage en applikasjon som lagrer *studenter* og *kurs* og det er sammenheng mellom dem.

Jeg følger i hovedsak prinsippene i Domain Driven Design (DDD)[4] som har følgende lag:



Evans modell

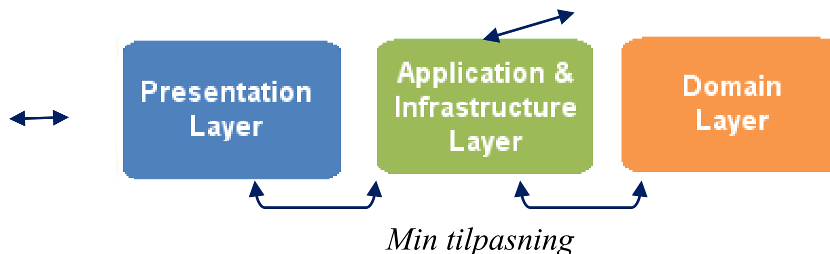
[3] "Samtidig" er ikke mulig på en tidsdelt maskin, men endringene kan være flettet så en endring ikke er ferdig når den settes på vent for en annen klients endring..

[4] Eric Evans, 2004, i boken "Domain Driven Design", se f.eks. http://www.methodsandtools.com/archive/archive.php?id=97

Applikasjonen skal da ha laginndeling, med separasjon mellom
1. presentasjonslaget (grenseobjekter som håndterer brukerne)
2. applikasjonslaget (kontrollobjekter som styrer oppdateringene)
3. domenelaget (entitetsobjektene som tar vare på data)
4. infrastrukturlaget (grenseobjekter som håndterer andre systemer f.eks. databaser og filer)

Det er bare presentasjonslaget (1) som tillates å kommunisere med brukeren og bare infrastrukturlaget (4) som får kommuniserer med andre systemer. Øvrige objekter kommuniserer seg imellom som vist med piler – de sender meldinger og får svar og kaster evt. feil bare til nærmeste lag.

Jeg har motforestillinger mot deler av denne modellen. Det er lagt opp til at domenelagets objekter selv håndterer persistensen (av seg selv) gjennom infrastrukturlaget. Det vil innebære at hvert enkelt objekt i domenelaget må passe på å lagre seg selv på ytre lager hver gang det endrer tilstand. Da må de alle ha operasjoner for det. Det vil gi en god del duplisering av kode. Jeg mener derfor det er enklere at applikasjonslaget – som kjenner til alle endringer av objektenes og systemets tilstand – også sikrer persistensen. Jeg vil derfor slå sammen applikasjonslaget og infrastrukturlaget. Mitt forslag til modell blir da slik:



Min tilpasning

Feil skal håndteres så snart det er mulig – feil som ikke kan håndteres kastes videre inntil de evt. når presentasjonslaget som varsler brukeren om feilen.

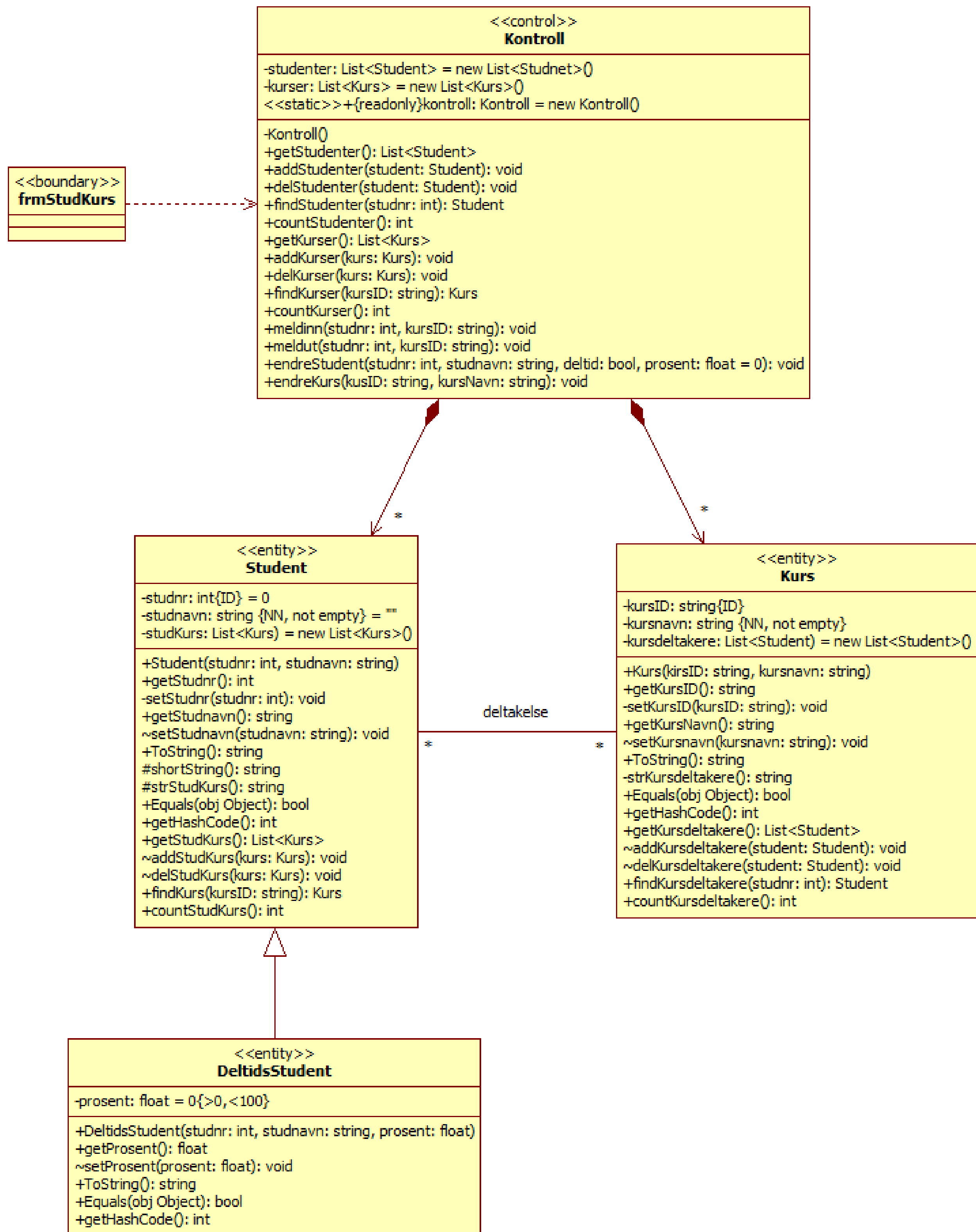
I programeksemplene benytter jeg C#.NET som kan lastes ned gratis fra Microsoft mot registrering[5]. C# er et mye brukt språk for applikasjoner til Microsoft Windows, det er objekt- og hendelsesorientert og i motsetning til Java er det svært enkelt å lage gode grensesnitt til brukeren. Programmeringsmiljøet (IDE) Visual Studio er etter min mening meget bra.

Jeg gjør alle attributter (variable) *private* så sant det er mulig, og lager tilgangsoperasjoner (tilgangsfunksjoner) for dem. Tilgangsoperasjoner for henting av attributtverdier kan vanligvis ha synlighet *public*, da de ikke kan endre objektets tilstand. Tilgangsoperasjoner for ny verdi, må beskyttes mer, fordi de vil endre objektets tilstand, og det skal bare objekter i applikasjonslaget (og ikke presentasjonslaget) ha lov til etter DDD-modellen.

[5] Hentes fra http://www.microsoft.com/express/Downloads/#2010-Visual-CS

Modell

Applikasjonen tar utgangspunkt i følgende klassediagram tegnet med UML-syntaks[6]:



Modell for eksemplet (komplett)

[6] En grei oversikt – som jeg ikke har kontrollert og følgelig ikke kan gå god for – finnes hos Wikipedia på http://en.wikipedia.org/wiki/Class_diagram

Skjemaobjektet frmStudKurs

frmStudkurs er det skjemaet som håndterer systemets bruker. Det har hendelsesoperasjoner som viser data og utfører de handlinger brukeren initierer gjennom mus og tastatur. Jeg har valgt å ikke spesifisere i modellen hvilke operasjoner dette er.

Entitetsklassene

Jeg har gitt alle entitetsklassene et ID-attributt. I objektorientert programmering er jo ikke det nødvendig – objektets plassering i RAM identifiserer det. Når objektene lagres på et ytre lager, har de imidlertid ikke lenger denne plasseringen, og neste gang de hentes til RAM vil de sannsynligvis få en annen plassering. For å sikre persistens av referansene mellom objektene, må de referere til hverandre også når de er lagret på ytre lager. Derfor må de få en entydig identifikator – en OID – som de andre objektene kan referere til. Det er også da mulig for persistensmekanismene å se om et objekt allerede er hentet til RAM eller om det må gjøres. Egentlig burde persistensmekanismene selv kunne generere en identifikator – og noen av dem gjør det også. Andre klarer ikke det, og da er det greit å tilby dem et attributt som er identifiserende for hvert enkelt objekt. Jeg benytter allikevel som vanlig i OOP, direkte referanser mellom objekter i RAM. Noen ganger – f.eks. i *find* – er det allikevel svært greit å ha en slik identifikator som utgangspunkt for å finne riktig objekt og returnere referansen til dette objektet.

Alle entitetsklassene skal ha en konstruktør. Jeg mener som prinsipp at klassens konstruktør(er) minst må ha med som argument alle de "kritiske" attributtverdiene, dvs. alle de som må gis eksplisitt verdi. Da sikrer man at ingen kan skape objekter som bryter mot klassens regler. Det kan gjerne lages flere konstruktører, når det er praktisk, men enklere er det å ha alle attributtene med som argument, men da gjøre de ikke-kritiske attributtene *optional* (hvis språket tillater det). I mitt eksempel er det gjort slik.

Noen persistensteknikker krever at det finnes en argumentfri konstruktør og delvis også at den skal være *public*. Jeg må da fravike mine prinsipper og innføre en slik konstruktør.

Enkelte persistensteknikker krever at objektene realiserer grensesnittet *Serializable*. Det endrer ingenting ved klassen eller objektene og krever ikke realisering av noen operasjoner, men gjør det klart for kompilatoren at programmereren faktisk har tenkt å tillate objektene å bli serialisert. Når det er nødvendig, legger jeg da til det.

Jeg har i utgangspunktet ikke laget destruktører. Det advares av mange mot å lage destruktør bl.a. fordi destruktøren kjøres av garbage gollector (gc) ved behov uten at vi har kontroll over det. Evt. feil som oppstår vil bli kastet til gc og får ingen effekt. Allikevel gjør mange unntak og mener at destruktør er aktuell hvis det er ressurser som må uttrykkelig frigis når objektet slettes. Microsoft nevner filer og grafikk som eksempler. Her kan det være aktuelt å lukke filer, databaser o.l.

Jeg legger ellers til de vanlige operasjonene *ToString*, *Equals* og *getHashCode*. For noen klasser legger jeg til ytterligere hjelpeoperasjoner, f.eks. *shortString* som studentobjektene selv bruker internt.

Jeg har merket alle attributtene privat. Det må derfor lages tilgangsoperasjoner for dem. Jeg ønsker at alle som har referanse til et objekt skal kunne *hente* objektets attributter. Det endrer jo ikke objekts tilstand på noen måte, og er derfor uten konsekvenser for persistensen. Jeg lager følgelig alle *get*-operasjonene public. Det gjør det også betydelig enklere å programmere at skjemaet

frmStudkurs viser dataene til brukeren – vi kan trygt la skjemaet få oppgitt referanser til entitetsobjekter.

For *set*-operasjonene gjør jeg den forutsetning at ID-attributter aldri skal kunne endres etter at objektet er opprettet. Det vil føre til svært mange problemer, da ID-attributtet typisk brukes som identifikator på ytre lager og av brukeren. Jeg gjør derfor *set*-operasjoner for ID-attributter *private*. De brukes kun av objektets konstruktør[7].

De andre *set*-operasjonene skal være tilgjengelig for kontrollobjektet, men ikke for andre. I noen språk kan vi da angi *friend*-objekter som får lov til å bruke operasjoner i klassen. I C# (og Java) er dette ikke mulig. I C# kan man ordne det på et vis ved å merke operasjonene *internal* og legge kontrollklassen og entitetsklassene i en egen *assembly*, dvs. slik at de kompileres særskilt. Operasjonene blir da bare synlige for andre objekter innen samme *assembly*. Siden det ikke finnes noe skjema i denne *assembly*, vil de i .NET kompileres til et klassebiliotek (*dll*-fil). I diagrammet ovenfor har jeg merket dem med ~ (som i UML egentlig betyr *package* hvilket er noenlunde det samme).

For enkle attributter (verdier og referanser til ett objekt) lager jeg bare én *get*- og én *set*-operasjon. Attributter som er samlinger (collections), gir jeg imidlertid fem operasjoner:

1. *get* henter hele samlingen (dvs. referansen til den)
2. *add* legger et nytt objekt til samlingen
3. *del* sletter et objekt fra samlingen
4. *find* tar ID-attributtet som argument og returnerer ett objekt fra samlingen (dvs. referansen til det)
5. *count* returnerer antallet i samlingen

Kontrollklassen

Klassen *Kontroll* er spesiell, da det er en såkalt *singleton*. Det skal ikke instansieres flere objekter av denne klassen. Vi vil jo sikre at alle tilstandsendringer går gjennom dette objektet slik at persistensen kan sikres. Hvis det ble instansiert flere slike kontrollobjekter, ville de kunne "ødelegge for hverandre". Det er minst to måter å gjøre dette på. En metode som jeg tror er ganske vanlig hos dem som programmerer med C++, er å gjøre klassen abstrakt og ha alle attributter og operasjoner som *static*. Det har bl.a. den fordelen at alle objekter i systemet kan referere til den og sende den meldinger vha klassenavnet. I Java er det mitt inntrykk at man foretrekker å skape objekter. Her viser jeg en måte å gjøre det på, samtidig som jeg sikrer at det ikke kan skapes mer enn ett objekt av kontrollklassen. Jeg lar klassen være konkret (ikke abstrakt). Da kan den i utgangspunktet instansieres. Jeg gjør imidlertid konstruktøren privat, så den kun kan brukes av klassen selv. Jeg skaper en *static, public, readonly* instans ved å angi følgende attributt:

```
static public readonly Kontroll kontroll = new Kontroll();
```

(At attributtet er *readonly* innebærer at objektet ikke kan endres etter at det har fått verdi gjennom initiering eller i en konstruktør[8].)

Dette attributtet, som altså er et Kontrollobjekt og er *static*, kan nås av alle gjennom referansen

```
Kontroll.kontroll
```

[7] Noen vil nok synes at dette er å gå for langt. Hvis man er sikker på at kun konstruktøren tilordner ID-attributtet, kan jo verdikontrollen ligge der. Ved å legge kontrollen i en egen *set*-metode, oppfyller jeg imidlertid et greit prinsipp og gjør programmet mer robust ved endringer.

[8] *const* kan ikke benyttes her, av årsaker som det fører for langt å komme inn på

I tillegg har jeg to samlinger (lister) med referanse til alle entitetsobjektene, med de vanlige tilgangsmetoder. Operasjonen som legger til en ny student har et studentobjekt som argument. Jeg planlegger å la skjemaobjektet skape et nytt studentobjekt (da må alle objektets attributter oppgis som parametre) og så få lagt dette til. Tilsvarende gjelder kursobjekter.

Videre lager jeg en operasjon som melder en student inn i et kurs og en som melder studenten ut av kurset igjen. Dessuten har jeg operasjoner som endrer et bestemt entitetsobjekt (alle de nye verdiene oppgis – identifikatoren kan ikke endres men oppgis så man kan finne riktig objekt å endre).

Persistens

Jeg tenker å hente alle objektene fra ytre lager når programmet starter, og så lagre dem igjen først når programmet avslutter. Det er ikke særlig trygt, da alle endringer i en sesjon kan mistes hvis sesjonen avsluttes ureglementert. Programeksemplene blir på den annen side enklere å følge, og enkelte persistensteknikker lagrer alt i én fil. Da blir det tungt å lagre alt hver gang ett objekt har endret tilstand.

Programeksemplene

Det blir nødvendig å endre modellen litt senere når jeg programmerer de forskjellige teknikkene. Det er fordi teknikken stiller bestemte krav til systemet.

Skjema

Skjemaet som brukes ser slik ut:

Som det fremgår, kan man endre og legge til en student og et kurs. Når en student, eller kurs er valgt i listeboksene, kan student/kurs også slettes. Dessuten kan man legge til og slette (men ikke endre) en assosiasjon (kalt "kobling" i dette skjemaet) mellom en student og et kurs. Alle endringer skal vises umiddelbart i de to listeboksene.

A: Applikasjonen uten persistens

Uten persistens er det systemet som brukes som eksempel uten verdi. Når et programeksempel uten persistens allikevel er laget, er det fordi det er utgangspunkt for alle programeksemplene *med* persistens. Programmet omgjøres etter behov så persistensen sikres på forskjellige måter.

Programeksempel – uten persistens

Applikasjonen er gjengitt i vedlegg A.

Jeg antar at koden er selvforklarende. Det eneste som det kanskje kan være greit å vite, er at presentasjonslaget (*frmStudKurs*) får liten betydning for persistens. Det er bare tatt med i vedlegget for å gjøre eksemplet komplett.

B: Lagring i flat, kommalimitert (csv) fil

Lagring/henting av objektenes tilstand med kommalimitert fil krever endringer i skjemaklassen og kontrollklassen. Kommalimitert fil er en meget enkel lagringsmåte. Den er direkte lesbar og kan importeres i mange andre programmer. Her er f.eks. dataene importert til Excel regneark:

A11

	A	B	C	D
1	KURS	INF10	C#	
2	KURS	INF20	VB	
3	KURS	INF30	OOAD	
4	STUDENT	15	Knut	
5	STUDENT	25	Olga	
6	STUDENT	35	Petter	
7	STUDKURS	15	INF10	
8	STUDKURS	15	INF20	
9	STUDKURS	25	INF10	
10				

Det er mulig for en bruker å editere filen med en editor.

Programmereren har full kontroll med hvordan lagringen skjer, hvor den skjer og filformatet. Det kan valgfritt programmeres med sikte på eksekveringshastighet, sikkerhet eller annet.

Programeksempel – lagring i flat, kommalimitert (csv) fil

Entitetsklassene (Student, Deltidsstudent og Kurs)

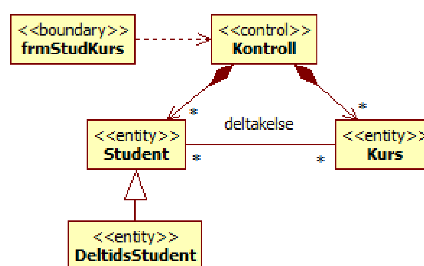
Persistens påvirker ikke disse klassene i det hele tatt.

Skjemaklassen

Hvis brukeren skal bestemme filnavnet for dataene, kan man bruke en *OpenFileDialog* og en *SaveFileDialog*. Dataene kan hentes ved oppstart, og lagres før avslutning, med kall til kontrollklassens metode *lesData(filnavn)* og *lagreData(filnavn)*.

Kontrollklassen

Kontrollklassen foretar lagringen og hentingen av entitetsobjektene.



Modell for eksemplet (gjentatt uten detaljer – komplett graf side 7)

Det er et mange-til-mange forhold mellom studenter og kurs, men ikke alle studenter er tilknyttet kurs og ikke alle kurs er tilknyttet studenter.

Man kunne da tenke seg å gå igjennom alle studenter og lagre dem og deres kurs. Da vil også sammenhengen mellom studenter og kurs komme med. Dessverre kan man ikke være sikker på å få med alle kurs på denne måten, da noen kurs er uten studenter. Det vil også skape vanskeligheter med å unngå at et kurs lagres flere ganger. Tilsvarende er det ingen løsning å gå igjennom alle kurs

og lagre dem sammen med deres studenter. Da kan man ikke være sikker på å få med alle studentene, da noen av dem er uten kurs, og det vil bli dubletter av studenter.

Man er følgelig nødt til å lagre alle studentene uten sine kurs, ved å gå igjennom hele listen *studenter* (som har referanse til alle studentene). Deretter gjøres tilsvarende for alle kurs i listen *kurser*. Det som da gjenstår, er sammenhengen mellom studenter og kurs. Her kan man gå igjennom alle kurs og lagre deres koblinger til studenter[9]. Dette tilsvarer logisk sett løsningen med mange-til-mange relasjoner i relasjonsdatabaser, der man lagrer sammenhengene i en egen (koblings-)tabell. På denne måten kommer alle koblingene med.
Jeg benevner hver linje i filen med hvilken objekttype dataene som er lagret på denne linjen tilhører. Det skal lette lesingen og gjenoppbygningen senere. Lagringen gjøres slik:

```
public void lagreData(string connString)
{
    //Lagrer alle data i CSV-fil
    //CSV-strengene bygges her så man slipper egen operasjon i objektene
    StreamWriter skriver = new StreamWriter(connString, false);
    foreach (Kurs kurs in kurser)
        skriver.WriteLine("KURS," + kurs.getKursID() + ","
            + kurs.getKursnavn());
    foreach (Student stud in studenter)
        if (stud.GetType() == typeof(Student))
        {
            skriver.WriteLine("STUDENT," + stud.getStudnr().ToString()
                + "," + stud.getStudnavn());
        }
        else
        {
            skriver.WriteLine("DELTID,"
                + ((Deltidsstudent)stud).getStudnr().ToString()
                + "," + ((Deltidsstudent)stud).getStudnavn() + ","
                + ((Deltidsstudent)stud).getProsent().ToString());
        }
    //Må enten gå igjennom alle studenter og deres kurs,
    //eller alle kurs og deres studenter
    foreach (Student stud in studenter)
        foreach (Kurs kurs in stud.getStudkurs())
            skriver.WriteLine("STUDKURS," + stud.getStudnr().ToString() + ","
                + kurs.getKursID());
    skriver.Close();
}
```

Det er bevisst at student- og kursobjektene lagres før sammenhengene. Ved lesing av filen, vil dermed alle student- og kursobjekter eksistere når sammenhengene mellom dem leses.

Filen som genereres kan f.eks. se slik ut:

```
KURS,INF10,C#
KURS,INF20,VB
KURS,INF30,OOAD
STUDENT,15,Knut
STUDENT,25,Olga
STUDENT,35,Petter
STUDKURS,15,INF10
STUDKURS,15,INF20
STUDKURS,25,INF10
```

[9] Alternativt kan man gå igjennom alle studentene og lagre deres kobling til kurs. De to løsningene gir samme resultat – bare rekkefølgen blir annerledes.

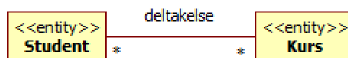
Lagringen kan kaste feil, som evt. vil bli håndtert av skjemaobjektet.
Når objektene bygges opp igjen "pakkes dataene ut" linje for linje, og et objekt skapes som har tilsvarende tilstand. Linjer som gjelder sammenhenger, fører til endringer i begge de to involverte objektene.

```
        public void lesData(String fil)
        {
            //Leser alle data fra CSV-fil
            StreamReader leser = new StreamReader(fil);
            String[] data; String linje;
            while (!leser.EndOfStream)
            {
                linje = leser.ReadLine();
                data = linje.Split(',');
                switch (data[0])
                {
                    case "STUDENT": //lag ny student
                        Student nystudent = new Student(Int32.Parse(data[1]), data[2]);
                        studenter.Add(nystudent);
                        break;
                    case "KURS": //lag nytt kurs
                        Kurs nyKurs = new Kurs(data[1], data[2]);
                        kurser.Add(nyKurs);
                        break;
                    case "STUDKURS": //sett ny relasjon student<->kurs
                        Student tmpStudent = findStudenter(Int32.Parse(data[1]));
                        Kurs tmpKurs = findKurser(data[2]);
                        tmpStudent.addStudkurs(tmpKurs);
                        tmpKurs.addKursdeltakere(tmpStudent);
                        break;
                    default:
                        throw new Exception("Feil i innfilen - kan ikke skape noe");
                }
            }
            leser.Close();
        }
```

Også lesingen kan gi feil som videresendes til skjemaobjektet.

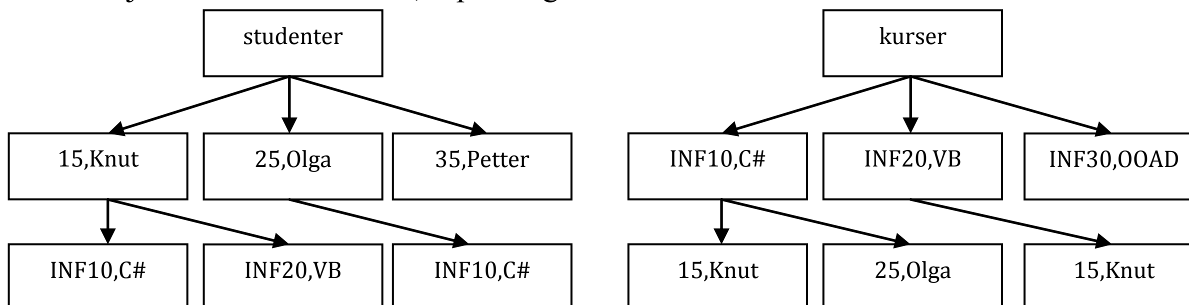
C: Lagring av serialiserte objekter (binær form)

De fleste objekter kan *serialiseres*. Det innebærer å gjøre om objektet til en bitstrøm som senere kan settes sammen igjen til et objekt. Prosessen kalles *serialisering* (serialization, deflating). Den motsatte prosessen – fra bitstrøm til objekt – kalles da *deserialisering* (deserialization, inflating). Mange språk, også C# og Java, har egne klasser som hjelper til med serialiseringen. Det gjør kodingen mye enklere.



Modell for eksemplet (gjentatt og forenklet uten detaljer – komplett graf side 7)

Når objekter serialiseres, tas objekter som det refereres til også med (*eager* strategi). I vårt tilfelle vil det innebære at serialisering av alle studentobjektene vil ta med de kursene som studentene henviser til. Fortsatt vil det da finnes kurs som ikke blir lagret, fordi de ikke har studenter. Hvis man da i tillegg serialiserer kursobjektene (for å få med alle), vil de ta med mange – men ikke alle – studentobjekter. De vil danne to, separate grafer:



Problemet er at når kursene lagres med studenter inkludert, blir studentene lagret som *nye objekter* selv om de er lagret fra før. Det er naturligvis ikke ønskelig. Det kan unngås ved uttrykkelig å implementere grensesnittet *ISerializable* med metoden *GetObjectData()* som brukes i serialiseringsprosessen. Man kan også merke attributter som man ikke vil ha med som *[NonSerialized]* (Java: *transient*). Det løser imidlertid ikke problemet her, for hvis referansene *kurs.kursdeltakere* og tilsvarende for studentobjektene merkes *[NonSerialized]*, så kommer sammenhengene mellom ikke objektene med i det hele tatt.

Løsningen på dette er å samle alle studentobjekter og kursobjekter i ett objekt, dvs. et *container* objekt. Når dette objektet serialiseres, kommer alt med. Serialiseringsobjektene er "smarte nok" til ikke å lagre studentobjekter/kursobjekter dobbelt. I eksemplet gjennomfører jeg en slik endring i kontrollobjektet der referansene til kurs- og studentobjektene ligger.

De ferdige klassene *BinaryFormatter* og *Stream* gjør hele arbeidet for oss, så programmeringsmessig er det svært enkelt. Vi kan også regne med at det blir riktig. Vi får imidlertid mindre kontroll over hva som skjer.

Det er en ulempe at det ikke er mulig å endre filen for å ta høyde for at et, enkeltobjekt er endret. Isteden må hele strukturen lagres pånytt. Hvis applikasjonen har mange objekter, vil det ta tid.

Det er ikke mulig i praksis å endre filens innhold manuelt. Den lar seg heller ikke enkelt importere til andre programmer.

Programeksempel – lagring i binær form

Slik lagring gjennomføres med *BinaryFormatter*.

BinaryFormatter stiller følgende krav:

1. Objekter som skal serialiseres, må merkes som serialiserbare
2. Det kan bare lagres én struktur eller objekt i hver fil – det som skal lagres sammen må derfor legges inn i én struktur eller ett objekt.

Entitetsklassene (Student, Deltidsstudent, Kurs)

Enitetsklassene skal serialiseres og må da merkes som serialiserbare med *[Serializable]*[10]. Klassen merkes helt enkelt slik:

```
[Serializable] //til serialisering binært - eneste nødvendige endring
public class Student
{
osv.
```

Også subklasser må merkes på denne måten (serialiserbarhet arves ikke i C#).

Andre endringer i enitetsklassene er ikke nødvendige.

Skjemaklassen

Det er kontrollklassen som foretar selve lagringen, så endringer i skjemaobjektet er unødvendig.

Kontrollklassen

Det kan bare serialiseres ett objekt/struktur i hver fil. Jeg skal lagre en liste med studenter og en liste med kurs. Da må jeg samle dem i et container-objekt. Her definerer jeg en struktur *DataStruct* som container. Også den må være serialiserbar.

```
[Serializable] //kreves av BinaryFormatter
private struct DataStruct //Container for de to listene, må være serialiserbar
{
    public List<Student> studenter;
    public List<Kurs> kurser;
    public DataStruct(List<Student> studenter, List<Kurs> kurser) //koblingene
    {
        this.studenter = studenter;
        this.kurser = kurser;
    }
}
private DataStruct data =
    new DataStruct(new List<Student>(), new List<Kurs>());
//Denne klassen gjøres som en singleton med bare én instans, her kalt "kontroll"
//Man får tilgang til den ved å skrive "Kontroll.kontroll"
static public readonly Kontroll kontroll = new Kontroll();
```

Ved å serialisere strukturen *data* kommer alt med.

Alle referanser til *studenter* og *kurser* må gjøres om til *data.studenter* osv. (*data.studenter.Add(student);* istedenfor bare *studenter.Add(student);*) Merk at selv om listene er

[10] Dette kalles i C# for et "klasseattributt" og regnes som en egenskap ved alle objekter i klassen. Hvis noen av attributtene ikke skal serialiseres, kan de hver for seg merkes *[NonSerialized]*. Hvis man selv vil skrive koden for serialiseringen, kan man isteden la klassen implementere grensesnittet *ISerializable* og man må da skrive kode for metoden *GetObjectData()*. I Java gjøres tilsvarende ved at klassen implementerer grensesnittet *ISerializable* som ikke inneholder noen metoder, så implementeringen er altså svært enkel der også.

gjort *public* (forenkler programmeringen) så er selve strukturen *data* merket *private*. Informasjonsskjulingen er derved fortsatt ivaretatt.

Lagringen håndteres med et objekt av klassen *BinaryFormatter* og skrives til fil gjennom et *Stream*-objekt. Slik kan det se ut:

```
public void lesData(string connString)
{
    //Les alle data serialisert binært
    Stream innstrøm = File.Open(connString, FileMode.Open);
    BinaryFormatter formatterer = new BinaryFormatter();
    data = (DataStruct)formatterer.Deserialize(innstrøm);
    innstrøm.Close();
}
public void lagreData(string connString)
{
    //Lagre alle data serialisert binært
    Stream utstrøm = File.Open(connString, FileMode.Create);
    BinaryFormatter formatterer = new BinaryFormatter();
    formatterer.Serialize(utstrøm, data);
    utstrøm.Close();
}
```

Som det fremgår, lagres alle objektene i strukturen *data* serialisert med én eneste setning:

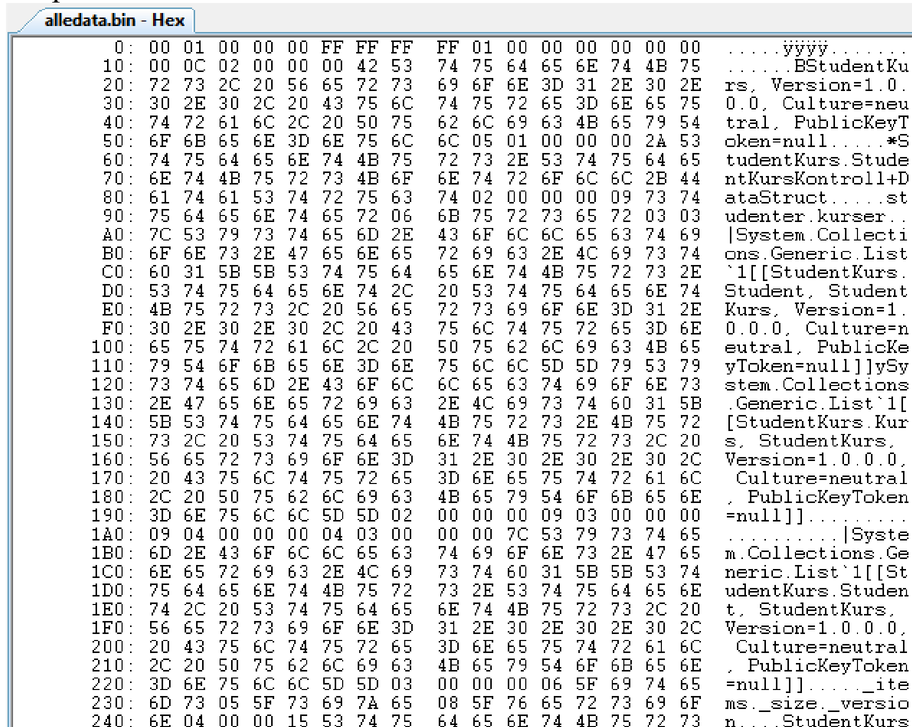
```
formatterer.Serialize(utstrøm, data);
```

Serialize tar én bytestrøm og ett objekt som parametre.

Tilsvarende bygges objektene igjen med en enkel setning. *Deserialize* returnerer ett, eneste objekt av klassen *Object* og derfor må resultatet typekastes til *DataStruct*:

```
data = (DataStruct)formatterer.Deserialize(innstrøm);
```

Den filen som skapes får binært innhold som det ikke er lett å få noe ut av. Her er en del av den:



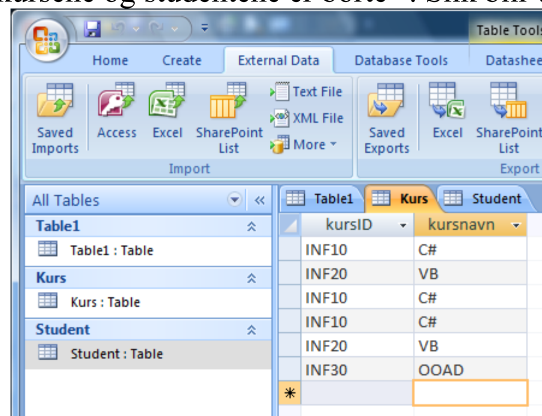
D: Lagring i XML-fil

Objekter kan også serialiseres til en XML-fil. Jeg skal se på to klasser som kan hjelpe til med serialisering og deserialisering, henholdsvis i kapittel D1 og D2.

Som kjent er XML et filformat som "organiserer data i en hierarkisk struktur. Formatet er et vanlig tekstformat, leselig for mennesker, der *merker*, eller *tagger*, gir informasjon om hva innholdet er."[11] Formatet egner seg bl.a. godt for overføring av data mellom systemer, da mange programmer støtter XML-formatet.

Det er mulig å bruke to ferdige klasser til lagring i XML, *XmlSerializer* og *DataContractSerializer*. Forskjellen på de to er at den første er ganske enkel å bruke, men den godtar ikke sirkelreferanser slik den andre gjør. Begge krever at alle egenskaper er *public* og det bryter med prinsippet om informasjonsskjuling. At det også kreves argumentfri konstruktør er imidlertid ingen stor ulempe. De kan bare serialisere ett objekt. Vi må følgelig samle alle de persistente objektene i ett, kunstig container-objekt for å få dette til.

Filen blir lesbar, og kan endres, av mennesker, men filen fra *DataContractSerializer* er vanskelig å lese pga sirkelreferansene. (Et utdrag av den filen som produseres, er gjengitt nedenfor.) Det er også mange andre programmer som kan anvende en XML-fil[12]. Et forsøk på å importere filen til Access gir imidlertid feil. Samme kurs kommer da med flere ganger (slik de også er lagret i XML-filen) og alle sammenhenger mellom kursene og studentene er borte[13]. Slik blir tabellen:



Videre kan man selv bestemme (med tillegg i koden) hvilke XML-tag'er som skal brukes.

[11] Wikipedia (http://no.wikipedia.org/wiki/XML) nov. 2011
[12] Excel kan imidlertid ikke lese fra XML-fil.
[13] Studentene kommer riktig med – hver student er lagret bare én gang i XML-filen.

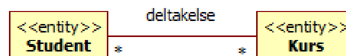
D1: Lagring i XML-fil med XmlSerializer

Til dette bruker man et objekt av klassen *XmlSerializer* (istedenfor *BinaryFormatter*) og et *Stream*-objekt.

XmlSerializer stiller følgende krav:

1. Attributtene må være *public*. Det ødelegger altså innkapslingen.
2. Den kan bare serialisere ett objekt (som *BinaryFormatter*) og det må være et *objekt* – den klarer altså ikke å serialisere en *struct*.
3. Klassene behøver ikke merkes *[Serializable]*.
4. Alle referanser blir også serialisert, men den håndterer ikke sirkelreferanser (det gir feilmelding når objektet oppdager det).

Vi har sirkelreferanser når objekt A refererer til objekt B som igjen refererer til A – evt. via et tredje objekt. Det er nettopp det vi har i mitt eksempel:



Modell for eksemplet (gjentatt og forenklet – komplett graf side 7)

Vi kan unngå sirkelreferansene ved å merke attributter *[XmlIgnore]*, f.eks. kan vi her merke *Kurs.kursdeltakere* slik. Det gir følgende XML-fil (utdrag):

```
<DataStruct>
 <studenter>
  <Student>
   <studnr>15</studnr>
   <studnavn>Knut</studnavn>
   <studkurs>
    <Kurs>
     <kursID>INF10</kursID>
     <kursnavn>C#</kursnavn>
    </Kurs>
    <Kurs>
     <kursID>INF20</kursID>
     <kursnavn>VB</kursnavn>
    </Kurs>
   </studkurs>
  </Student>
```

... *osv student for student. Deretter*

```
 <kurser>
  <Kurs>
   <kursID>INF10</kursID>
   <kursnavn>C#</kursnavn>
  </Kurs>
```

... *osv for resten av kursene*

```
 </kurser>
</DataStruct>
```

Prinsippet er tegnet ovenfor under binær serialisering. Som det fremgår, er alle kursene til student nr 15 kommet med sammen med studenten. Samme kurs kan komme med flere ganger (når flere studenter tar samme kurs). I tillegg er de samme kursene kommet med under *kurser* og der er det noen flere (kurs som ikke har studenter). Studentene blir imidlertid *ikke* lagt inn i listen *kurser* sammen med kursen der, da de der er merket *[XmlIgnore]*.

Når dette leses inn igjen, vil det bli *flere* eksemplarer av mange kurs, f.eks. skapes kurs INF10 to ganger. De tolkes som to forskjellige objekter og legges to forskjellige steder i RAM. Dermed blir det ikke enkelt å gjenoppbygge referansene fra kurs til studenter, men umulig er det ikke. Det er gjort i programeksemplet og vist nedenfor (side 7 ff).

Hvis alle referanser er obligatoriske skaper dette ingen problemer. Hvis f.eks. en person har referanse til flere biler og *alle* biler har referanse til en eier, kan man la være å lagre bilene særskilt – alle kommer med som referanse fra en eier.

Programeksempel – lagring i XML-fil med XmlSerializer

Entitetsklassene

Alle attributtene må være *public*.

Videre må klassene ha en argumentfri konstruktør. Den kan være *private* og derved beskyttet mot bruk fra andre objekter, som fortsatt kan bruke konstruktøren med argumenter. For klassen Student går ikke dette, for da blir den ikke tilgjengelig for subklassen *Deltidsstudent*. Jeg setter den derfor til *protected*. I de andre entitetsklassene setter jeg den *private*.

```
public class Student
{
    public int studnr = 0;
    public string studnavn = "";
    public  List<Kurs> studkurs = new List<Kurs>();

    public Student(int studnr, string studnavn)
    {
        setStudnr(studnr);
        setStudnavn(studnavn);
    }
    protected Student() //kreves av XmlSerializer
        //(protected for å være tilgjengelig for subklasser)
    {
    }
```

For Deltidsstudent ser den argumentfrie konstruktøren slik ut:

```
    private Deltidsstudent(): base () //kreves av XmlSerializer
    {
    }
```

I klassen Kurs må listen *kursdeltakere* ikke tas med i serialiseringen – den gir sirkelreferanser. Det unngår jeg ved å merke den *[XmlIgnore]*:

```
    [XmlIgnore]
    public List<Student> kursdeltakere = new List<Student>();
```

Ellers er entitetsklassene uforandret.

Skjemaklassen

Skjemaklassen er det ikke nødvendig å endre.

Kontrollklassen

Her skal strukturen *data* som er en *DataStruct* serialiseres. Da må den ha argumentfri konstruktør som kan være *private*. Ettersom *struct* ikke kan ha argumentfri konstruktør (alle feltene i strukturen må gis verdi når den skapes) må jeg gjøre den om til en *klasse*.

Videre må klassen være *public* og attributtene det samme:

```
public class DataStruct //Container for de to listene, må være public
{
    public List<Student> studenter;
    public List<Kurs> kurser;

    public DataStruct(List<Student> studenter, List<Kurs> kurser)
    {
        this.studenter = studenter;
        this.kurser = kurser;
    }
    private DataStruct() { } //XML serializer krever parametfri konstruktør
    {
    }
}
```

Videre må det naturligvis skrives ny *lagreData* der jeg helt enkelt bytter *BinaryFormatter* med *XmlSerializer*:

```
public void lagreData(string connString)
{
    //Lagre alle data i XML-fil
    Stream utstrøm = File.Open(connString, FileMode.Create);
    XmlSerializer XMLformatterer
        = new XmlSerializer(
            typeof(DataStruct), new Type[] { typeof(Deltidsstudent) });
        //subklasser må nevnes eksplisitt
    XMLformatterer.Serialize(utstrøm, data);
    utstrøm.Close();
}
```

Legg merke til at konstruktøren for XmlSerializer må ha kjennskap til subklasser. De legges til i en array *Type[]*.

Metoden *lesData* må endres tilsvarende:

```
public void lesData(string connString)
{
    //Les alle data fra XML-fil
    Stream innstrøm = File.Open(connString, FileMode.Open);
    XmlSerializer XMLformatterer
        = new XmlSerializer(
            typeof(DataStruct),
            new Type[] { typeof(Deltidsstudent) });
            //subklasser må nevnes eksplisitt
    data = (DataStruct)XMLformatterer.Deserialize(innstrøm);
    innstrøm.Close();
```

Når denne lesingen er gjennomført, har studentene fått referanser til sine kurs, men det er laget egne kursobjekter til hver student. I tillegg er alle kursobjektene laget og lagt inn i listen *kurser*. Sistnevnte har imidlertid ingen referanser til sine studenter fordi listen *kursdeltakere* ble ignorert ved serialiseringen. Studentene refererer heller ikke til dem, men til kopier av dem. Dette må det gjøres noe med:

```
            //Studentene har fått sine kurs, men de er kloner av kursene i listen kurser
            //(1) Bytt ut referansen i studentenes liste studkurs med referanse til
            //tilsvarende kurs i listen kurser.
            //(2) Legg deretter studenten til i kursets liste kursdeltakere
            foreach (Student stud in data.studenter)
                for (int i = 0; i < stud.studkurs.Count; i++) //kan ikke bruke foreach
                {
                    stud.studkurs[i] = findKurser(stud.studkurs[i].getKursID()); //(1)
                    stud.studkurs[i].addKursdeltakere(stud); //(2)
                }
            //Da skulle alle referanser være på plass begge veier
        }
```

Som man ser, er dette tungvint og tidkrevende og kan fort gjøres feil i litt mer komplekse systemer.

D2: Lagring i XML-fil med DataContractSerializer

Ettersom jeg anser problemet med sirkelreferanser som alvorlig, er det naturlig å se om det ikke finnes en annen klasse som kan håndtere det. En slik klasse er *DataContractSerializer*. For at den skal håndtere sirkelreferanser, må riktig konstruktør benyttes, men da slipper man de "triksene" som måtte til med *XmlSerializer*. Også den kan bare serialisere ett objekt (pr fil).

DataContractSeriliazer stiller følgende krav:

1. Attributtene må være *public*. Det ødelegger altså innkapslingen.
2. Den kan bare serialisere ett objekt og det må være et *objekt* – den klarer altså ikke å serialisere en *struct*.
3. Klassene behøver ikke merkes *[Serializable]*.
4. Det kreves referanse til biblioteket *System.Runtime.Serialization.dll*.

Programeksempel – lagring i XMLfil med DataContractSerializer

Entitetsklassene

[XmlIgnore] kan nå fjernes fra attributtet *kursdeltakere* i klassen *Kurs* (den har heller ingen effekt nå).

Ellers er ingen endringer nødvendig i entitetsklassene (fortsatt er attributtene *public* og jeg serialiserer objektet *data*).

Skjemaklassen

Ingen endringer er nødvendig.

Kontrollklassen

Det må selvsagt gjøres endringer i *lesData* og *lagreData*, der *DataContractSerializer* erstatter *XmlSerializer*:

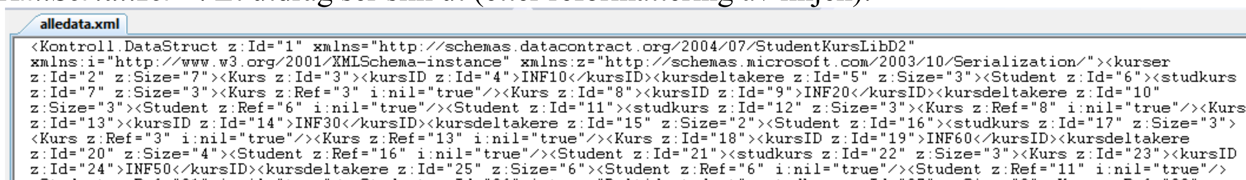
```
public void lagreData(string connString)
{
    //Lagre alle data i XML-fil med DataContractSerializer
    Stream utstrøm = File.Open(connString, FileMode.Create);
    DataContractSerializer XMLformatterer
        = new DataContractSerializer(typeof(DataStruct),
            new Type[] { typeof(Deltidsstudent) }, //subtype - må være med
            int.MaxValue,
            false,
            true, //lag egne objektID, unngår probleme med sirkelreferanser
            null);
    XMLformatterer.WriteObject(utstrøm, data);
    utstrøm.Close();
}
public void lesData(string connString)
{
    //Les alle data fra XML-fil
    Stream innstrøm = File.Open(connString, FileMode.Open);
    DataContractSerializer XMLformatterer
        = new DataContractSerializer(typeof(DataStruct),
            new Type[] { typeof(Deltidsstudent) }, //subtype - må være med
            int.MaxValue,
            false,
            true, //lag egne objektID, unngår probleme med sirkelreferanser
            null);
    data = (DataStruct)XMLformatterer.ReadObject(innstrøm);
    innstrøm.Close();
}
```

Konstruktørene for DataContractSerializer har mange former[14]. Her bruker jeg følgende syntaks[15]:

```
public DataContractSerializer(
    Type type, //type for objektet som skal serialiseres
    IEnumerable<Type> knownTypes, //andre typer, f.eks. subtyper
    int maxItemsInObjectGraph, //maksimalt antall objekter
    bool ignoreExtensionDataObject, //se bort fra "Type Extensions"
    bool preserveObjectReferences, //Sentral: Unngår problemer med sirkelreferanser
    IDataContractSurrogate dataContractSurrogate //spesielle definisjoner for XML-filen
    )
```

Denne er jo litt kompleks, men til gjengjeld har den ingen problemer med sirkelreferanser.

XML-filen som produseres, har bare én, lang linje og er ikke så enkel å tolke som den som lages av *XmlSerializer*[16]. Et utdrag ser slik ut (etter reformattering av linjen):


alledata.xml


```
<Kontroll.DataStruct z:Id="1" xmlns="http://schemas.datacontract.org/2004/07/StudentKursLibD2"
xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"><kurser
z:Id="2" z:Size="7"><Kurs z:Id="3"><kursID z:Id="4">INF10</kursID><kursdeltakere z:Id="5" z:Size="3"><Student z:Id="6"><studkurs
z:Id="7" z:Size="3"><Kurs z:Ref="3" i:nil="true"/><Kurs z:Id="8"><kursID z:Id="9">INF20</kursID><kursdeltakere z:Id="10"
z:Size="3"><Student z:Ref="6" i:nil="true"/><Student z:Id="11"><studkurs z:Id="12" z:Size="3"><Kurs z:Ref="8" i:nil="true"/><Kurs
z:Id="13"><kursID z:Id="14">INF30</kursID><kursdeltakere z:Id="15" z:Size="2"><Student z:Id="16"><studkurs z:Id="17" z:Size="3">
<Kurs z:Ref="3" i:nil="true"/><Kurs z:Ref="13" i:nil="true"/><Kurs z:Id="18"><kursID z:Id="19">INF60</kursID><kursdeltakere
z:Id="20" z:Size="4"><Student z:Ref="16" i:nil="true"/><Student z:Id="21"><studkurs z:Id="22" z:Size="3"><Kurs z:Id="23"><kursID
z:Id="24">INF50</kursID><kursdeltakere z:Id="25" z:Size="6"><Student z:Ref="6" i:nil="true"/><Student z:Ref="11" i:nil="true"/>
```


Et hovedpoeng er at hvert objekt (inkludert objekter som f.eks. *string*) gis et unikt nummer som identifikator- en OID (*object identifier*). Når det da er nødvendig å henvise til et objekt som allerede er kommet med, henvises bare til det unike nummeret. Dermed unngås at et objekt lagres flere ganger og alle referanser kommer med.

[14] Se http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datacontractserializer.aspx
[15] Se http://msdn.microsoft.com/en-us/library/aa344262.aspx
[16] Filen lar seg ikke uten videre importere, hverken til Access eller (meningsfylt) til Excel

E: Lagring i en relasjonsdatabase

Lagringsmetodene som er brukt ovenfor, er alle i flate, enkeltstående filer. Det kan være greit når dataene bare skal brukes av én klientapplikasjon av gangen. I praksis skal ofte flere applikasjoner dele dataene og bruke dem samtidig, og da blir slike filer upraktiske. Man må jo sørge for at klientapplikasjonene ødelegger for hverandre, f.eks. ved at flere applikasjoner samtidig endrer objektenes tilstand og lagrer igjen. Da vil den som lagrer sist "vinne" og overskrive endringer gjort av de andre ("lost update").

I flerbrukersystemer har man følgelig behov for å styre filbruken. Man kan i prinsippet låse filen når én klient vil gjennomføre endringer, lagre endringene og så frigi filen igjen. Det vil føre til mye venting for de andre klientene. I praksis blir det uholdbart.

Videre vil man gjerne sikre dataene mot uautorisert tilgang. Det kan ordnes ved å sikre at bare visse brukere, visse IP-adresser eller visse maskiner får tilgang til filen. Det blir tungvint.

Den vanlige løsningen på slike problemer, er å lagre dataene i en database. Man bruker et ferdig databasehåndteringssystem (DBMS) som har funksjonalitet for å kontrollere tilgangen til data, organisere samtidighet, kontrollere lovligheten av endringer, gjenopprette data ved feil, logge alle endringer osv. Det finnes forskjellige databasetyper, men de aller fleste benytter nå relasjonsdatabaser. Her skal jeg drøfte og vise hvordan det kan gjøres. Senere skal jeg vise alternativer.

Jeg bruker her samme strategi som i foregående eksempler, ved at jeg lagrer *alle* objektene til slutt. Jeg burde jo heller lagret endringer etter hvert som de skjedde. Slik det foregår her, blir ingen endringer lagret før brukeren klikker knappen "Lagre", og hvis systemet stopp under lagringen, kan man miste endringer.

Tilsvarende burde jeg hente ett og ett objekt fra databasen etter hvert som jeg trengte det til noe.

Jeg tror allikevel at det jeg har gjort her, demonstrerer hvor tungvint (og vanskelig) det er å bruke relasjonsdatabase til å lagre objekter. Først må man foreta en vanskelig mapping, deretter må man skrive mye kode selv for å bygge objekter og lagre dem. Det blir fort problemer med brudd på beskrankninger, men de vil gi feil som selvsagt bør fanges (det gjør ikke jeg i eksemplet ovenfor).

Når man ser hvor kompliserte dette er for et svært lite og enkelt eksempel, synes jeg det er overraskende at dette nok er den vanligste lagringsmåten. Antakelsen bygger på at relasjonsdatabaser er den klart mest brukte databasetypen, samtidig som objektorientert programmering blir stadig vanligere.

Programmeringen blir mye enklere, men krever mer minne, hvis man knytter applikasjonen til en database og lagrer dataene i applikasjonen i et *DataSet*-objekt. *DataSet*-objektet fremstår da som en "virtuell database" og knyttes til databasen med et *SqlDataAdapter*-objekt. Det viser jeg ikke her.

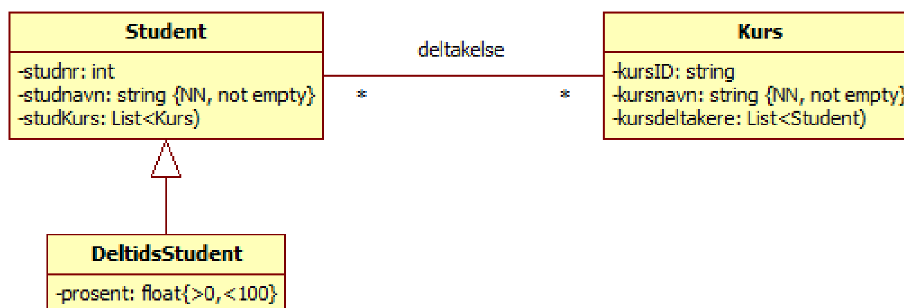
En mulighet er også å lage sitt eget bibliotek for konverteringen. Det er fortsatt en god del jobb å sette opp mappingen mellom objektenes felt og databasetabellenes kolonner, men når det først er gjort, vinner jeg mye ved programmering av lagring (oppdatering, sletting, innsetting) i databasen. Allikevel er dette ganske tungvint og krever mye spesialprogrammering. Det kan fort gå galt.

Uansett vil det gi en god del run time overhead å gjøre konvertere objekter fra/til poster i relasjonsdatabasen. Ved transaksjonsintense applikasjoner blir dette merkbart.

Mapping

Relasjonsdatabaser lagrer bare data – ikke objekter – og objektorienterte systemer benytter mekanismer som ikke finns i relasjonsdatabaser. Noen eksempler er at relasjonsdatabaser ikke håndterer innkapsling, informasjonsskjuling, arv, abstraksjon eller mange-til-mange relasjoner. På den annen side bruker relasjonsdatabasene mekanismer som vi ikke har i objektorientert programmering, bl.a. primærnøkkel, fremmednøkkel og krav om at alle verdier skal være atomære. Dessuten er datatypene i de to systemene ikke like – vi kan ikke engang vite at et heltall representeres likt i de to systemene og heller ikke om de heter det samme. Når man vil lagre avbildninger av objekter i en relasjonsdatabase, må man følgelig endre strukturen. Det objektorienterte systemet må "mappes" til relasjonsdatabasen.

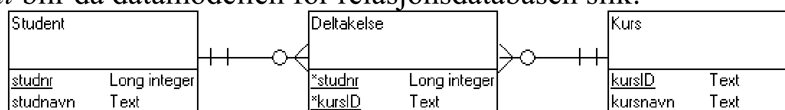
Entitetsklassene i mitt eksempel har følgende struktur (operasjonene er skjult – de kan uansett ikke lagres i relasjonsdatabasen, men det er det heller ikke behov for):



Modell for eksemplet (gjentatt og forenklet – komplett graf side 7)

Det er i utgangspunktet rimelig å innføre en tabell for hver klasse. Mange-til-mange forholdet mellom Student og Kurs "entitetiseres" til en tabell der hver rad har referanse til akkurat én student og akkurat ett kurs. Denne tabellen erstatter listene *studkurs* og *kursdeltakere*. Relasjonsdatabasen krever at tabellene har en primærnøkkel som identifiserer hver rad, mens identiteten av et objekt i RAM er gitt ved dets RAM-adresse. Vi må følgelig velge oss, eller innføre, et attributt som kan være primærnøkkel i databasen. Her er vi så heldige å ha *studnr* og *kursID* som begge er forskjellige for hvert objekt. De kan altså brukes som primærnøkler her og som fremmednøkkel der det er behov for å referere mellom poster. Vi måtte ellers innføre et slikt attributt/kolonne.

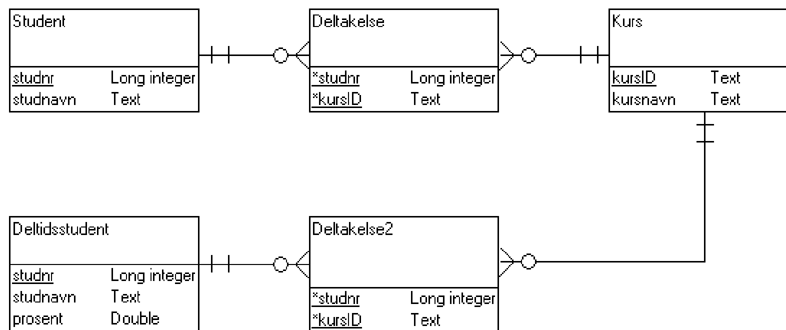
Uten Deltidsstudent blir da datamodellen for relasjonsdatabasen slik:



Det gjenstår da å mappe arvehierarkiet *Student - Deltidsstudent*. Det finnes minst fire måter, kalt *strategier*, å gjøre det på – her skal jeg vise tre av dem[17]:

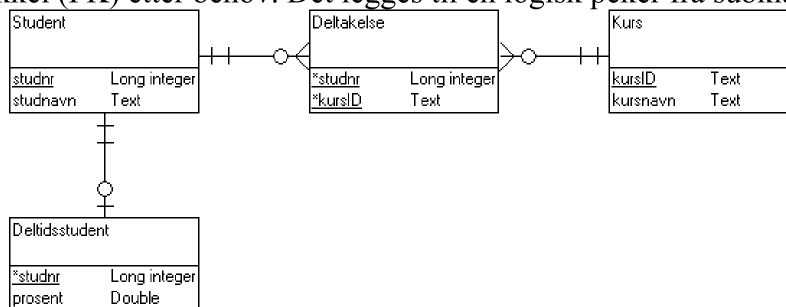
[17] Den fjerde – "generisk mapping" – er svært generell og kompleks og sjelden aktuell når man mapper manuelt.

"Horisontal mapping" (partisjonering)
Denne strategien innebærer at bare de *konkrete* klassene mappes med alle attributter – også de som er arvet – til hver sin tabell og at primærnøkler og fremmednøkler innføres etter behov. Slik vil det se ut:



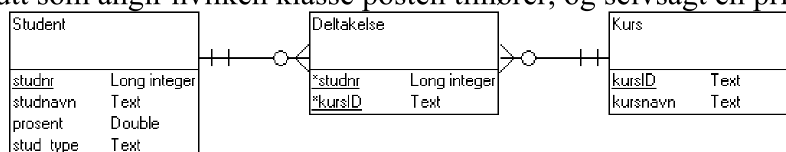
Det hadde vært fint om tabellen *Deltakelse* kunne brukes både til å koble Student og Deltidsstudent til Kurs, men det går dessverre ikke (*Deltakelse.studnr* er fremmednøkkel og man må angi hvilken tabell den refererer til). Denne modellen vil derfor bli ganske tungvint i bruk[18].

"Vertikal mapping" (separasjon)
Denne strategien innebærer at alle klasser (også abstrakte klasser) mappes til hver sin tabell. Man innfører primærnøkkel (PK) etter behov. Det legges til en logisk peker fra subklasse til metaklasse.



Her unngår vi dubleringen av tabellen *Deltakelse*. Når en *Deltidsstudent* skal lagres, legges navnet i tabellen *Student*, *prosent* legges i tabellen *Deltidsstudent* og *studnr* legges begge steder. Hver rad i *Deltakelse* viser til en *Student* – som kan vise seg (hvis det finnes en rad i *Deltidsstudent* med samme *studnr*) å være en deltidsstudent. Dette gir tunge og kompliserte spørringer.

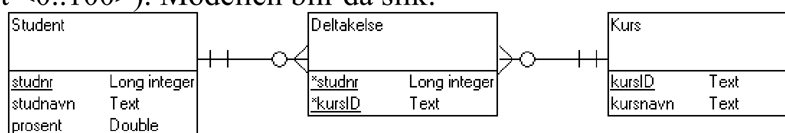
"Filtrert mapping" (absorpsjon)
Denne strategien innebærer at alle tabellene i et arvehierarki slås sammen til én tabell. I tillegg kreves da en attributt som angir hvilken klasse posten tilhører, og selvsagt en primærnøkkel.



Dette er modellert "direkte etter reglene", men modellen kan forenkles. For det første er det her bare to studenttyper. Da kan attributtet *stud_type* erstattes av en Boolsk verdi, f.eks. *heltid* som er sann hvis dette er en heltidsstudent. Vi kan ytterligere forenkle, hvis vi innfører en regel om at alle

[18] Det kan være fristende å slå sammen de to tabellene for deltakelse. Det går dessverre ikke, for da må både referansen til student og til deltidsstudent være med i primærnøkkelen. Det er ikke tillatt da én av dem alltid vil være *null*. Med en annen (innført) primærnøkkel, er det ikke mulig å sikre mot dobbelt registrering av én deltakelse.

heltidsstudenter – og bare dem – har *prosent* lik 100 (det er ellers angitt at deltidsstudenter skal ha *prosent* i intervallet <0..100>). Modellen blir da slik:



Når en heltidsstudent lagres, sikrer vi at *prosent* settes til 100. Når vi henter en student, sjekker vi verdien av *prosent* og avgjør da om det er et *Student*-objekt eller et *Deltidsstudent*-objekt som skal skapes.

Det er den siste datamodellen jeg velger å bruke her, da den åpenbart gir enklest lagring/henting.

Normalisering av datamodellen

Datamodeller bør normaliseres til minst fjerde normalform (4NF). Deretter kan man eventuelt bevisst ødelegge normaliseringen for å gjøre noe enklere, øke hastigheten, spare plass e.l. Jeg normaliserer selv etter følgende prinsipper:

Step A – "Databasesyn": Gjør om modellen slik at

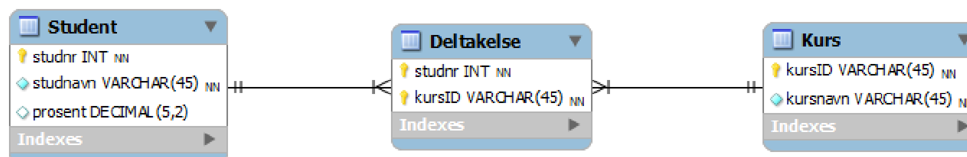
1. alle entitetstyper har minimal identifikator, og
2. alle relasjonstyper er eksplisitt gitt ved fremmednøkkel, og
3. alle attributtyper er atomære

Step B – Fjerde normalform: Gjør om modellen slik at alle determinander er kandidat-nøkkel

I modellen jeg valgte ovenfor, er alle disse kravene oppfylt og det er ikke nødvendig å gjøre noe for å normalisere modellen.

Skape databasen

I programeksemplene bruker jeg MySQL lokalt på min maskin (den kan naturligvis ellers gjerne befinne seg på en tjener). I MySQL lager jeg databasen *Studkurs* slik:



Jeg skulle gjerne ha laget en beskrankning (constraint) som sikret at *kursnavn* og *studnavn* ikke settes til tom streng (*Not Null* sikrer bare mot *null*) og at *prosent* er i intervallet <0..100], men MySQL støtter ikke slike beskrankninger. Man kunne laget en trigger, men oppnår lite med det. Det er jo også slik at når objekter skapes i applikasjonen min, så kontrolleres dette. Siden databasen bare er en avbildning av objektene, vil denne feilen uansett ikke kunne forekomme ved lagring. Jeg lar det derfor ligge her. Ulempen er først og fremst at hvis noen går inn i databasen direkte eller med andre applikasjoner, vil de kunne sette ulovlige verdier på de nevnte attributtene. Det kan gi feil når min applikasjon leser dataene og gjenskaper objektene.

Programeksempel E1: Lagring i relasjonsdatabase – programmerer alt selv

Jeg antar her at databasen er skapt som forklart ovenfor.

Relasjonsdatabaser stiller følgende krav:

1. Objektene må ha en identifikator.
2. Det må lages en tilpasset relasjonsdatabase. Det må være samsvar mellom det objektorienterte systemet og relasjonsdatabasen, dvs. det må mappes en sammenheng

mellom dem.
3. Det må skrives mye kode for å konvertere objektene til poster i tabeller og motsatt.

Applikasjonen

Jeg tenker å knytte applikasjonen til database med ADO-teknologi. Jeg kunne også benyttet ODBC, men den er eldre. For å kunne bruke ADO mot MySQL må jeg hente ned et passende klassebibliotek. MySQL tilbyr flere biblioteker – det vi trenger her er "ADO.NET Driver for MySQL (Connector/NET)" som hentes fra MySQLs hjemmesider[19]. Dette er en zip-fil som inneholder installasjonsfil for driveren. Den må kjøres og vil skape/registrere biblioteket MySql.Data.MySqlClient[20].

Det må legges til en referanse til dette biblioteket i programmet, og en referanse til System.Data.

Entitetsklassene

Jeg gjør ingen endringer her.

Skjemaklassen

I konstruktøren skaper jeg en *connectionstring* (deklarert som egenskap for skjemaet) som inneholder det nødvendige for å få kontakt med databasen:

```
this.Show();
connectionString =
    "Server=localhost;" +
    "Database=Studkurs;" +
    "Uid=Knut;" +
    "Pwd=Tunk;" +
    "Connect Timeout=30;" +
    "CharSet=utf8;";
kontroll.lesData(connectionString);
ryddSkjema();
```

Her burde naturligvis brukernavn og passord vært hentet fra brukeren og ikke hardkodet, men det er trivielt så jeg utelater det her.

Videre blir det endringer i *butLagre_Click*, som nå blir meget enkel:

```
private void butLagre_Click(object sender, EventArgs e)
{
    kontroll.lagreData(connectionString);
    ryddSkjema();
}
```

Ellers er skjemaet uforandret.

Kontrollklassen

I kontrollklassen blir det ganske mye endringer, når jeg nå skal **lese fra en database**.

```
public void lesData(String fil)
{
    string studError = ""; // til feilmeldinger for studenter
    string kursError = ""; //ditto kurs
    string deltakelseError = ""; //ditto deltakelser
```

[19] http://www.mysql.com/products/connector/

[20] Mer detaljer om MySqlClient finnes mange steder på Internett. Et av de bedre er http://www.csharp-station.com/Tutorials/AdoDotNet/Lesson01.aspx

Jeg starter med å knytte applikasjonen til databasen. Da trenger jeg en *connection* basert på den oppgitte *connectionString* og den må åpnes. Videre trenger jeg et kommandoobjekt og et leserobjekt:

```
MySqlConnection dbStudkurs = new MySqlConnection(connectionString);
dbStudkurs.Open();
MySqlCommand dbKommando;
MySqlDataReader dbLeser;
```

Med disse kan jeg nå lese databasen, ved å sette riktig kommando og skape leseren. Leseren har ingen tilgjengelig konstruktør, men skapes ved å kalle kommandoens *ExecuteReader* som returnerer en leser knyttet til denne kommandoen:

```
//Les og bygg alle kursobjekter
dbLeser.Close(); //leseren må lukkes for den reåpnes
dbKommando =
    new MySqlCommand("select * from Kurs order by kursID asc;", dbStudkurs);
dbLeser = dbKommando.ExecuteReader();
```

Det er nå klart for å lese datatabellen *Student* rad for rad og skape objekter av dem. Leserens *Read*-metode flytter til neste rad og inneholder deretter dataene i raden. Dataene hentes med *GetFloat*, *GetString* osv. Jeg skiller mellom *Student*-objekter og *Deltidsstudenter* ved å sjekke prosent (100% betyr *Student*):

```
while (dbLeser.Read())
{
    try
    {
        Student nyStudent;
        if (dbLeser.GetFloat(2) == 100)
            nyStudent = new Student(dbLeser.GetInt32(0),
                dbLeser.GetString(1));
        else
            nyStudent =
                new Deltidsstudent(dbLeser.GetInt32(0), dbLeser.GetString(1),
                    dbLeser.GetFloat(2));
        addStudenter(nyStudent);
    }
    catch (Exception ex)
    {
        studError += ex.Message;
    }
}
```

Helt tilsvarende henter jeg alle kurs, men her slipper jeg å skille mellom heltid og deltid.

Referansene (assosiasjonene) mellom studenter og kurs, finnes i tabellen *Deltakelse*. Hver rad gir oppdatering av én student og ett kurs, ved at referansen legges til:

```
//Les og bygg alle deltakelser
dbLeser.Close();
dbKommando =
    new MySqlCommand("select * from Deltakelse order by studnr,kursID asc;",
        dbStudkurs);
dbLeser = dbKommando.ExecuteReader();

while (dbLeser.Read())
{
    try //Bygg opp feilmelding hvis noe skjer under lesingen
    {
        int studnr = dbLeser.GetInt32(0);
        string kursID = dbLeser.GetString(1);
        findStudenter(studnr).addStudkurs(findKurser(kursID));
        findKurser(kursID).addKursdeltakere(findStudenter(studnr));
    }
    catch (Exception ex)
    {
        deltakelseError += ex.Message;
    }
}
```

Det gjenstår bare å kaste evt. oppsamlede feil, så er alle objektene hentet fra databasen og gjenopprettet i RAM.

Også når jeg skal **skrive til databasen** må jeg gjøre ganske mye.

Jeg starter med å knytte applikasjonen til databasen. Da trenger jeg en *connection* basert på den oppgitte *connectionString* og den må åpnes. Videre trenger jeg et kommandoobjekt (for lagring trenger jeg ikke noe eget "skriveobjekt").

Hvert objekt blir utgangspunkt for én insert-setning som kjøres med *ExecuteNonQuery*. Jeg må gjøre forskjell på *Student*-objekter og *Deltidsstudent*-objekter og jeg legger til "on duplicate key update" i tilfelle objektet allerede er lagret – da blir det erstattet:

```
foreach (Student stud in getStudenter())
{
    if (stud.GetType() == typeof(Student)) //heltidsstudent
        dbKommando.CommandText =
            "insert into Student set studnr=" + stud.getStudnr().ToString()
            + ",studnavn='" + stud.getStudnavn()
            + "',prosent=default on duplicate key update studnavn='"
            + stud.getStudnavn()
            + "',prosent=default;";
    else //deltidsstudent
        dbKommando.CommandText =
            "insert into Student set studnr=" + stud.getStudnr().ToString()
            + ",studnavn='" + stud.getStudnavn()
            + "',prosent=" + ((Deltidsstudent)stud).getProsent()
            + " on duplicate key update studnavn='" + stud.getStudnavn()
            + "',prosent=" + ((Deltidsstudent)stud).getProsent();
    dbKommando.ExecuteNonQuery();
}
```

Lagring av kurs skjer helt analogt, men uten noe skille mellom meta- og subklasser.

Da er det klart for lagring av alle assosiasjoner mellom objektene. De skal lagres i tabellen *Deltakelse* og det er det samme om jeg tar utgangspunkt i studentobjektene eller kursobjektene, da de viser til hverandre. Jeg benytter "replace" for å unngå problemer med dubletter. "Replace" er en insert, men endres til update hvis raden allerede finnes. Jeg kan ikke bruke insert med "on duplicate key update" her, fordi begge kolonnene er med i nøkkelen, så hvis nøklene kolliderer, er det ingenting å oppdatere. Et alternativ kunne være å ignorere feil, men det kan være problematisk hvis andre skrankefeil oppstår – da vil jo også de testene bli undertrykket.

```
foreach (Student stud in getStudenter())
    foreach (Kurs kurs in stud.getStudkurs())
    {
        dbKommando.CommandText =
            "replace into Deltakelse set studnr=" + stud.getStudnr() +
            ",kursID='" + kurs.getKursID() + "';";
        dbKommando.ExecuteNonQuery();
    }
```

Det gjenstår bare å lukke forbindelsen, så er lagringen gjennomført.

Programeksempel E2: Lagring i relasjonsdatabase med et hjelpeobjekt

For å avhjelpe litt de problemene foregående eksempel avslørte, har jeg laget en klasse som kan bistå med overgangen fra en rad hentet i databasen til objekt og fra objekt til SQL DML-setning (insert, delete, update og replace).

MySql.Data.Conversion

Dette er et bibliotek jeg har laget selv. Det tilbyr to objekter men applikasjonsprogrammereren bruker bare ett av dem, nemlig klassen *MySqlConverter*. *MySqlConverter* har følgende konstruktør:

```
public MySqlConverter(Type myType, string myTablename)
```

Man oppgir en *type* som angir hvilken klasse det skal konverteres til/fra og et *tabellnavn* som objektene i klassen lagres i, f.eks.

```
MySqlConverter studkonverter = new MySqlConverter(typeof(Student), "Student");
```

der objekter av klassen *Student* knyttes til databasetabellen *Student*. Det er bare mulig å angi én type og én tabell.

Når konvertererobjektet er skapt, bruker man metoden

```
public void addMapping(string fieldName, string columnName, bool key)
```

til å "mappe" ett og ett felt i objektet (av klassen *Student*) til én og én kolonne (i databasetabellen *Student*). Hvis denne kolonnen er en (del av) primærnøkkelen i tabellen, settes *key* til *true*. Minst en kolonne må være med i nøkkelen.

Her mapper jeg f.eks. alle felt/kolonner for *Student*:

```
studkonverter.addMapping("studnr", "studnr", true);
studkonverter.addMapping("studnavn", "studnavn", false);
studkonverter.addMapping("studkurs", null, false);
studkonverter.addMapping(null, "prosent", false);
```

Det er mulig å angi at et felt i objektene (her *studkurs*) ikke har noen tilsvarende kolonne i tabellen og motsatt (her *prosent* i tabellen).

Alternativt kan man først bygge opp en liste med Mapping-objekter, og så legge listen til i konvertererobjektet med

```
public void addMappingRange(List<MySqlMapping> mappinglist)
```

Konvertererobjektet er da ferdig "satt opp" og kan konvertere med følgende metoder:

1. `public object ToObject(IDataRecord arguments)`
   Skaper et objekt av en rad hentet fra databasen.
2. `public string ToInsertString(object insertObject)`
   Gjør om et object til en insert-setning.
3. `public string ToDeleteString(object deleteObject)`
   Gjør om et object til en delete-setning.
4. `public string ToUpdateString(object updateObject)`
   Gjør om et objekt til en update-setning.
5. `public string ToReplaceString(object replaceObject)`
   Gjør om et objekt til en replace-setning.

ToObject returner et objekt av Object-klassen – det må konverteres med "kasting" til riktig type. De tre andre, som genererer SQL DML-setninger, må få et objekt av samme type som ble oppgitt da konvertererobjektet ble skapt.

Programeksempel E3: Lagring i relasjonsdatabase med Entity Framework o.a. verktøy

Det finnes noen verktøy som kan bidra til å mappe objektorienterte systemer til relasjonsdatabaser og gjøre det mye enklere å lagre/hente data[21]. Disse er av minst tre typer:

1. De tilbyr verktøy som genererer en XML-fil som beskriver mappingen. Utgangspunktet er enten programmet (databasen kan skapes fra generert sql-skript) eller databasen (skaper klasser tilsvarende databasen).[22] En interessant mulighet er Entity Framework[23] som tilbyr begge de to metodene.
2. De tilbyr klassebibliotek som gir persistens gjennom arv[24].
3. De tilbyr modelleringsverktøy, som – etter at man har skapt en konseptuell modell – genererer både XML-filen og sql-skript med DDL (Entity Framwork – se tidl. ref).

Man må passé nøye på hvilket verktøy man bestemmer seg for, da mange av dem stiller krav til bestemte databaser eller programmeringsspråk.

[21] Wikipedia har en grei oversikt på http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software#.NET
[22] F.eks. Object Mapper .NET på http://www.objectmapper.net/overview.php
[23] Se f.eks. http://en.wikipedia.org/wiki/ADO.NET_Entity_Framework
[24] F.eks. Database Objects .NET på http://www.hisystems.com.au/databaseobjects/

F: Lagring i objektdatabase (db4o)

I det foregående har jeg diskutert og vist hvordan objekter kan lagres i en annen form – CSV-fil, binært, som XML-fil og som poster i en relasjonsdatabase. De første tre teknikkene egner seg dårlig i flerbrukersystemer, fordi de lagrer i filer som ikke er lett for flere å dele. De er imidlertid relativt enkle å programmere. Relasjonsdatabaser, på den annen side, er enkle å dele men medfører mye planlegging og programmering av konverteringen mellom objektorientert form og poster i tabeller. Man kan følgelig ønske seg et system som både egner seg for flerbrukermiljøer og som ikke medfører mye komplisert konvertering. Objektorienterte flerbrukerdatabaser kan kanskje være en slik teknologi[25].

Det er laget flere slike objektorienterte databaser, kalt *objektdatabaser*[26]. Det er også laget forslag til standard spørrespråk for objektdatabaser (OCL). Man så for seg en utvidelse av SQL til å håndtere objekter istedenfor poster. Dette har ikke slått an og prosjektet er nå nedlagt. Prosjektet har allikevel hatt stor innflytelse på noen av objektdatabasene. Db4o bruker ikke noe fra OCL.

Her skal jeg benytte programmet db4o (fra db4objects) som er fri programvare under GNU GPL lisens[27]. Ddb4objects Community drives nå av Versant (som også har en kommersiell objektdatabase), og er svært populær. Programmet har eksistert omtrent siden år 2000. Det er for tiden versjon 8 som brukes.

Db4o kommer i to utgaver: Det kan brukes i énbrukerversjon, slik vi gjør her (selv om vi later som det er klient/tjener) men det finnes også i en ekte klient/tjener versjon der tjeneren lytter på TCP/IP. Man må da lage et enkelt, lite program som starter db4o på tjenermaskinen. Denne versjonen tilbyr også transaksjonshåndtering, samtidighet (gjennom optimistisk og pessimistisk låsing) osv. ACID gjennomføres fullt ut og det finnes mange søkemuligheter. Jeg viser ikke en klient/tjener flerbrukerløsning her, men skal vise søkemulighetene.

Db4o er objektorientert. Grafen av objekter i RAM lagres som tilsvarende graf. Det er ingen konvertering, mapping eller annet. Man behøver ikke fortelle db4o om strukturen i grafen – det finner den selv i programmet ditt. Altså: Det er ingen "impedance mismatch" – vi programmerer alt objektorientert og med full kompilatorkontroll.

Det er svært enkelt å komme i gang med bruken (men selvsagt noe mer komplekst for full utnyttelse). Det har liten "footprint" og egner seg for "små RAM" f.eks. mobiler (den finnes for Java dvs. Android). Det er også raskt takket være sin SODA-teknologi. Dessuten sparer man mye run time ressurser ved at man slipper å konvertere og å mappe til/fra databasen.

Det kan sies å være en ulempe at det ikke er noe databasehåndteringssystem inkludert. Det betyr at man f.eks. ikke får noen adgangskontroll, recovery, integritetskontroll osv. Dette må programmereren lage selv. Db4objects tilbyr Object Manager Enterprise for direkte adgang til objektdatabasen, søking osv.

[25] Selv om programmereren kanskje ikke ser konverteringen, må den allikevel gjøres – det er ikke mulig å lagre et objekt i internminne som det er. Bl.a. må det nødvendigvis over på en seriell form for å kunne lagres på et sekvensielt, ytre lager. Referanser mellom objekter må også nødvendigvis konverteres, siden ytre lager ikke har noen minneadresser det kan refereres til.

[26] For meg er det litt rart at slike baser kalles "objektdatabaser" – jeg ville kalt dem "objektbaser" – men det er vanlig, så da holder jeg meg også til det.

[27] Det finnes også en kommersiell lisens, så db4o er "dual licensed".

Prinsippene for db4os virkemåte

Prinsippet for biblioteket, er at man skaper et *databaseobjekt* i programmet. Dette objektet håndterer databasen. Det finnes ikke noe Management System som det gjør i relasjonsdatabaser. All logikk, algoritmer, integritetskontroll, adgangskontroll, samtidighetshåndtering, transaksjonshåndtering osv. ligger altså på programsiden. Noe må man programmere selv, men det meste tilbys automatisert av det nevnte databaseobjektet.

Objektene lagres i en enkelt fil som man angir navnet på når objektdatabasen åpnes. Strukturen i denne filen er programmereren uvedkommende – det er databaseobjektet som selv håndterer den. Vi vet jo at objektene serialiseres på en eller annen måte, men behøver ikke tenke på hvordan det skjer og heller ikke hvordan filen er strukturert. Vi vet også at databaseobjektet legger til en unik objekt-id (OID) for hvert objekt som lagres, og selv holder orden på referansene mellom objektene med en logisk referanse ("fremmednøkkel"). Fra programmererens side fremstår det som databaseobjektet lagrer objektene i samme graf som de utgjør i internminnet. Den ekstra OID er intern for db4o og programmereren får aldri se den.

Databaseobjektet er av typen *IObjectContainer*. Det sørger for det første for lagring og henting av objekter fra/til internminnet på anmodning. All konvertering gjøres av databaseobjektet uten at vi behøver å programmere noe av det.

Videre "mapper" objektet sammenhengen mellom objekter i internminnet og OID i objektdatabasen. Db4o databaseobjektet holder selv orden på en "mapping" mellom RAM-adresse og OID. Vi kan se på det som en form for tabell (jeg vet ikke hvordan det er implementert, men det er *raskt*). I den ene kolonnen ligger da RAM-adressen for objekter som er hentet fra ytre lager – de sies å være "aktivert". I den andre kolonnen ligger OID. Denne tabellen er intern i databaseobjektet og opprettes når databaseobjektet "åpner objektdatabasen". Den slettes når databaseobjektet "lukker databasen". Ved neste åpning opprettes tabellen igjen *på nytt*. Dette innebærer at man må åpne databasen én gang og holde den åpen til siste slutt. Hvis ikke vil databaseobjektet gå surr i mappingen, og man får gjerne dubletter i databasen og/eller i RAM.

Virkemåten for denne mappingen kan kort forklares slik:

1. Når et objekt **aktiveres** (hentes til RAM) er det databaseobjektet som leser filen og skaper objektet av riktig type og laster det med data. Databaseobjektet vet altså hvor i RAM det havnet og det noteres i mappingtabellen. Hvis det blir aktuelt å aktivere det igjen – f.eks. fordi et søk returnerer det – sjekker databaseobjektet om dette objektet (med denne OID) allerede er aktivert. I så fall skapes det ikke pånytt, men RAM-adressen gjenbrukes.
2. Ved **lagring** av et objekt vet databaseobjektet også om objektet allerede er lagret og i så fall med hvilken OID. Dermed kan databaseobjektet *oppdatere* et objekt som allerede er i objektdatabasen. Hvis databaseobjektet ikke finner noen slik mapping, oppfatter det objektet som nytt, tilordner en ny OID og lagrer det som *nytt objekt* i objektdatabasen.
3. Ved **sletting** skjer tilsvarende oppslag før objektet slettes fra objektdatabasen (for å slette et objekt i objektdatabasen, må det først hentes til RAM og blir derved mappet).

Når man vil slette et objekt i RAM, derefererer man det. Når det ikke lenger finnes referanser til objekt blir det ikke mulig å finne det, og Garbage Collector (gc) vil etter hvert sette RAM ledig der objektet lå. Et interessant spørsmål da er hvordan Garbage Collector kan fjerne et slikt objekt, når databaseobjektet refererer til det (i sin mappingtabell)? Svaret er at mappingtabellen har såkalt "weak reference" som Garbage Collector ikke tar hensyn til[28].

[28] En grei forklaring i http://www.switchonthecode.com/tutorials/csharp-tutorial-weak-references

Når ett objekt A **hentes**, vil det jo ha *referanser* til andre objekter, f.eks. B. Da er spørsmål: Skal B også hentes? Husk at referansen fra A til B ikke er meningsfull uten at B ligger i RAM og har en RAM-adresse det kan refereres til. Hvis vi venter med å hente B, vil referansen settes til *null* – da bruker man en "lazy loading strategi". Hvis vi henter B, og kanskje da også C som B refererer til osv., bruker vi "eager loading". Hvor langt lenkene skal følges, kalles *activation depth* og standard er fem nivåer. Ofte betyr det at hele grafen (alle objektene) i praksis hentes, så det kan bli for mye – vi kan sette antall nivåer selv. Databaseobjektet passer selv på – ved å bruke mappingtabellen – at objekter som er hentet tidligere ikke hentes igjen. Det blir følgelig ingen "dubletter" i RAM. Man må allikevel passe på, for hvis man følger referanser i mer enn dybde 5, vil man støte på *nulls*. Da må man uttrykkelig "aktivere" referanser, dvs. hente dem til RAM.

Når et objekt A **lagres**, oppstår samme diskusjon. Hvis A har referanse til objekt B, bør da også objekt B lagres samtidig? Eller generelt: Hvor langt bør lenkene følges ved lagring? Dette kalles *update depth* og standard er ett nivå for *oppdatering* av objekter (objekt A finnes i objektdatabasen fra før) og ubegrenset for *nye* objekter (objekt A finnes ikke i objektdatabasen fra før). Det kan ofte bli for lite å lagre bare ett nivå, for ofte er også objekt B endret. Dette må programmereren vurdere og evt. endre.

Db4o lagrer bare *objekter*. *Klasser* kan ikke lagres og *static* attributter lagres følgelig heller ikke. Videre har db4o hatt problemer med collections. Det hevdes nå å være løst, men jeg får fortsatt problemer (det skyldes gjerne min manglende forståelse), så i programeksemplet legger jeg opp til å unngå det.

Vi må også huske på å lukke databasen – det anbefales i alle fall av Microsoft og av Db4objects – men jeg tror egentlig det er unødvendig. Lukkingen inkluderer automatisk *commit* og må gjøres helt tilslutt. Jeg anvender da en passende destruktør.

Det er ingen forbindelse, lytting eller annet mellom objektene i RAM og databaseobjektet. Databaseobjektet kan følgelig ikke vite om endringer av objektenes tilstand. Det innebærer at vi må programmere lagring, oppdatering og henting. I sin enkleste form, har databaseobjektet følgende metoder:

1. **Open()/Close()** åpner og lukker objektdatabasen. Mappingtabellen skapes/slettes også.
2. **Store()** lagrer et angitt objekt til objektdatabasen.
3. **Delete()** sletter et angitt objekt i objektdatabasen.
4. **Commit()/Rollback()** kommitterer/aborterer den løpende transaksjonen. En ny transaksjon startes umiddelbart. Alle endringer er transparente inntil kommittering.
5. **Query()/QueryByExample()** gir avansert og enkel søking i objektdatabasen.
6. **Activate()/Deactivate()** henter alle referanser i objektet til angitt dybde, evt. setter alle referanser til *null*. Det er først og fremst aktuelt ved *lazy loading*.

For flerbrukersystemer utvides denne listen med låsing, oppfrisking av et hentet objekt osv.

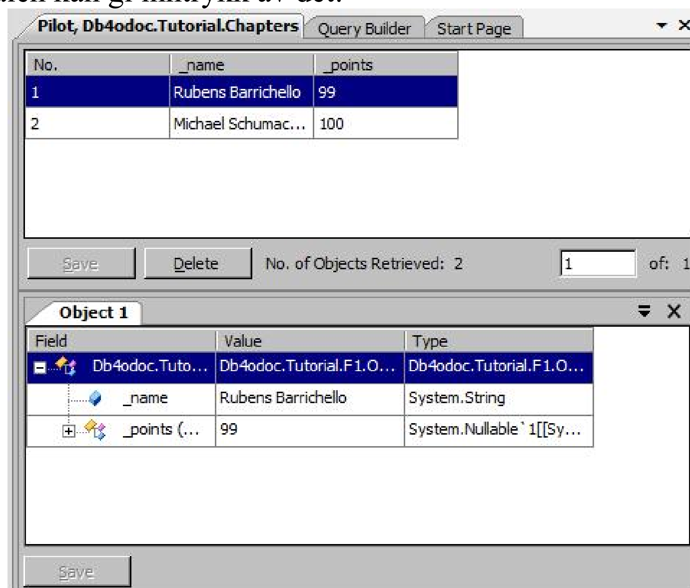
Når man søker i databasen, returneres alltid en samling med objekter. Selv om vi vet – fordi vi f.eks. søker med en ID – at det returneres maksimalt ett objekt, må vi bruke et slikt søk. Det finnes ingen søkemetode som returnerer en skalar (ett eneste objekt).

I relasjonsdatabaser er det vanlig å bygge indekser i forskjellige former, sortere dataene og annet. Med db4o kan man også skape indekser. Db4objects anbefaler selv at det bare gjøres for databaser med minst ti tusen objekter (av samme klasse), da hastighetsøkningen ellers er minimal. Db40 har

et eget *IndexDiagnostic*-objekt som kan hjelpe en databaseadministrator til å finne ut hvilke indekser som er lønnsomme.

I den mappingtabellen som *IObjectContainer*-objektet skaper og ajourholder, brukes hashing og indekser, så gjenfinning av det som er mappet der skal være så rask som mulig.

Det er heller ikke mulig å gå inn i objektdatabasen på andre måter, slik man kan med en relasjonsdatabase – all adgang må skje gjennom et program. Noen vil riktignok se på det som en fordel, men man mister muligheten til å generere/kjøre sql-skript. For db4o finnes programmet Object Manager Enterprise som installeres som en del av Visual Studio. Jeg har ikke forsøkt det, men et bilde fra manualen kan gi inntrykk av det:



Verktøyet gjør det også enkelt å bygge spørringer. Det kan også lages program for ad hoc spørringer mot objektdatabasen – eksempler finnes på Internett.

Programeksempel F1: Lagring med db4objects

Db4o stiller ingen krav til systemet.

Man trenger biblioteket DB4objects.DB4o.dll[29] for enklere, enbrukerapplikasjoner slik jeg demonstrerer her. Programmet installeres og jeg må referere til biblioteket i min applikasjon. Deretter er jeg klar til å programmere. Jeg bruker samme eksempel som ovenfor, selv om det viser litt lite av mulighetene i db4o.

Entitetsklassene

Jeg gjør ingen endringer i entitetsklassene.

Skjemaklassen

Det er ikke lenger bruk for å spørre brukeren om lagring når skjemaet lukkes – jeg lagrer alle endringer med en gang de er gjennomført i RAM.

Kontrollklassen

I tidligere eksempler benyttet jeg for enkelhets skyld en liste for objektene. Nå bruker jeg isteden databasen – den holdes oppdatert hele tiden.

[29] Lastes gratis ned fra http://www.db4o.com/community/qdownload.aspx?file=net40.msi

Deretter kan jeg deklarere et *IObjectContainer*-objekt. Det er dette objektet som håndterer objektdatabasen – man kan velge å se på det som selve databasen.

```
        private IObjectContainer objBase;
```

Objektdatabasen bør åpnes bare én gang og holdes åpen så lenge applikasjonen kjører. Jeg har lagt opp til åpningen skal skje fra skjemaklassen, så derfor skriver jeg metoden *åpneBasen*. Jeg setter noen parametre for demonstrasjonens skyld:

```
        public void åpneBasen(string connString)
        {
            Db4objects.Db4o.Config.IEmbeddedConfiguration minConfig
              = Db4oEmbedded.NewConfiguration();
            minConfig.Common.ActivationDepth = 5; //default ved henting
            minConfig.Common.UpdateDepth = 5;
                //OBS! Default er 1 og da lagres bare objektet selv og ikke objekter
                //som det referes til. Nye objekter lagres alltid fullt ut.
            objBase = Db4oEmbedded.OpenFile(minConfig,connString); //åpner med minConfig
        }
```

Tilsvarende bør objektdatabasen lukkes før avslutning. Jeg velger å gjøre det i destruktøren:

```
        ~Kontroll()
        {
            if (objBase!=null) objBase.Close();
        }
```

Metoder for å **hente/lagre alle objekter** kan fjernes – her holdes databasen à jour hele tiden.

Når vi skal **hente objektene**, må de "søkes" på en slik måte at alle returneres. Det kan søkes på fire, forskjellige måter: Query by example (QBE), native queries, LINQ queries og SODA queries. Db4objects selv anbefaler native queries, som gir alle muligheter og skrives direkte i C#. Jeg har brukt det enkelte steder ovenfor og bruker det også her[30]. Jeg viser flere muligheter i et eget programeksempel senere (side Programeksempel F2: Søking med db4objects44).

Merk at siden objektene – både student- og kursobjektene inneholder referanser, så blir også de hentet. Det er ingen separat lagring av referansene.

Når objekter **slettes fra, eller endres i, RAM**, vil db4objects ikke oppdage det. Man må følgelig aktivt gjøre tilsvarende endring i objektdatabasen, f.eks. i dette programeksemplet:

`objBase.Store(kurs);`	Lagrer nytt eller endret kurs. Nye lagres i full dybde, mens endrede bare oppdateres i dybden som ble angitt med *UpdateDepth* da basen ble åpnet.
`objBase.Ext().Store(kurs, 2);`	Som *objBase.Store* ovenfor, men her omgjøres *objBase* til et *IExtObjectContainer*-objekt og da kan man eksplisitt angi dybde.

`foreach (Student s in`
`        kurs.getKursdeltakere())`
`{`
	Sletter et kurs. Først må alle referanser til kurset slettes hos de deltakende studentene

[30] Jeg støter her på en liten utfordring: Native query returnerer den abstrakte datatypen *IList<object>* mens jeg har deklarert *getStudenter* og *getKurser* som *List*. Det gjør jeg helt enkelt om til *IList* uten at skjemaobjektet må endres av den grunn – der brukes den bare i *foreach*-strukturer uten problem. Forskjellen på en *IList* og en *List* er at førstnevnte bare henter OID uten å instansiere objektene, de blir først instansiert når de refereres.

`s.delStudkurs(kurs);`
`objBase.Store(s);`
`}`
`objBase.Delete(kurs);`
	og de endrede studentene lagres. Deretter slettes kurset.

`return objBase.Query<Kurs>().Count;`	Finner antall kurs.
`return objBase.Query<Kurs>();`	Henter en *IList* med alle kurs.

`public Kurs findKurser(string kursID)`
`{`
`IList<Kurs> kurser =`
`objBase.Query<Kurs>(k =>`
`k.getKursID()==kursID);`
`if (kurser.Count == 0) return null;`
`return kurser[0];`
`}`
	Finner et bestemt kurs. Returnerer *null* hvis ingen ble funnet, ellers returneres det som ble funnet (vi vet at det er maksimalt ett kurs med denne *kursID*). Merk at hvis ingen blir funnet, så vil *kurser[0]* innebære at databasen skal gjøre om OID til et objekt, og det gir feilmelding. når OID er *null*.

`stud.addStudkurs(kurs);`
`kurs.addKursdeltakere(stud);`
`objBase.Store(kurs);`
`objBase.Store(stud);`
	Legger en student til som deltaker på et kurs. Når de to involverte objektene er endret, blir begge lagret. Sletting av en kursdeltakelse gjøres tilsvarende.

Tilsvarende gjøres det for studenter.

Programeksempel F2: Søking med db4objects

Jeg skal her, med et eget eksempel, vise hvordan man kan gjenfinne objekter i en objektdatabase.

Jeg minner om at db4o primært lagrer objektene som en graf. Objektene refererer til hverandre med den interne OID. På den måten lagres også assosiasjonene på samme måte som i RAM, men uavhengig av objektenes RAM-adresser. Hvis det finnes en indeks som db4o kan utnytte på en lønnsom måte, så gjør den det uten at programmereren behøver å angi det.

Det er fire måter å søke i objektdatabasen på med db4o. Alle returnerer en samling:

1. **Query by Example (QBE)** der man gir db4o en prototype. Alle objekter som er "like" prototypen etter visse regler (som forklares nedenfor) returneres. Dette er enkelt for nybegynner, men mulighetene er sterkt begrenset, og det kan bli tungt og tregt. I dette eksemplet søker jeg etter alle kurs der kursID (første argument i konstruktøren) kan være hva som helst, men kursnavnet (det andre argumentet i konstruktøren) skal være "Databaser":
   ```
   Kurs proto = new Kurs(null, "Databaser");
   IObjectSet result = objBase.QueryByExample(proto);
   ```
2. **Native Query (NQ)** der spørringen skrives i vertsspråket og returnerer true/false. Koden i spørringen kjøres mot hvert objekt i objektdatabasen og resultatet true angir at objektet skal være med. Dette er noe vanskeligere, men mulighetene er bare begrenset av vertsspråket. Her gjør jeg det samme som under punkt 1:
   ```
   IList<Kurs> kurser = objBase.Query<Kurs>(delegate(Kurs kurs)
      {
         return kurs.getKursnavn() == "Databaser";
      });
   ```
   Selv foretrekker jeg en annen form som gjør det samme:
   ```
   IList<Kurs> kurser = objBase.Query<Kurs>
          (k => k.getKursnavn() == "Databaser");
   ```
3. **Language Integrated Query (LINQ)** er skapt av Microsoft og en del av .NET. LINQ likner ganske mye på SQL-spørringer, og er relativt enkelt å lære for dem som kjenner SQL. Eksemplet viser vel hvor meget det likner på SQL:
   ```
   IEnumerable<Kurs> kurser =
   ```

```
from Kurs k in objBase
where k.getKursnavn() == "Databaser"
select k;
```

4. **Simple Object Database Access (SODA)** følger referansene fra objekt til objekt og til egenskaper (felt) i objektet. Det kan legges beskrankninger på referansene. SODA utnytter altså at objektdatabasen er lagret som en graf og den er derfor effektiv. Av denne grunn blir LINQ-spørringer oversatt til SODA før de eksekveres. SODA kan imidlertid være vanskelig både å lære og å bruke.

```
IQuery query = objBase.Query();
query.Constrain(typeof(Kurs));
query.Descend("kursNavn").Constrain("Databaser");
IObjectSet result = query.Execute();
```

Alle metodene ovenfor gir samme resultat i de angitte eksemplene, nemlig alle kurs som har navnet "Databaser", men det er allikevel forskjeller på dem. Det ser vi gjennom andre eksempler i et programeksempel.

Oppsett av programeksemplet

Jeg har omtrent de samme entitetsklassene som tidligere: Student, Deltidsstudent og Kurs. Det er bare gjort mindre tilpasninger som følge av krav som særlig QBE stiller til klassene. De gjennomfører full informasjonsskjuling, men QBE krever argumentfri konstruktør, hvilket jeg ikke er begeistret før (det er diskutert tidligere).

Videre har jeg laget en ny kontrollklasse fra bunnen av, tilpasset dette programeksemplet, og en helt ny skjemaklasse. I kontrollklassen heter objektdatabasen (et *IObjectContainer*-objekt) objBase som før. Jeg tilknytter db4o-biblioteker etter behov og henviser til dem med *uses*-klausul.

QBE

For å benytte QBE, må klassene ha argumentfri konstruktør som skaper objekter uten initialiserte felter. Man kan således ikke sikre at objekter opprettes i en veldefinert, akseptabel tilstand.

QBE trenger et prototypeobjekt, der de feltene har verdi som man vil stille krav til. Det betyr at det er umulig å søke etter objekter med verdi 0 for tall, false for Boolske variable, tom streng o.l. for QBE vil anta at det innebærer at det ikke er stilt krav til feltet. QBE sammenlikner hvert objekt med prototypen og velger ut dem som har feltverdier lik dem som er angitt. Alle angitte felter må tilfredsstille likheten – det betyr at det gjennomføres en AND-operasjon mellom dem. Det er ikke mulig å angi jokertegn, det finnes ikke "like", "contains" eller andre spesifikasjoner. QBE er med andre ord ganske enkel. Til gjengjeld er det lett å lære.

Spørringen må sammenlikne alle objekter i objektdatabasen (av prototypeobjektets klasse) med prototypeobjektet. Det innebærer som regel at objektene må hentes til internminnet og det er tungt og tregt. Implementerte indekser kan imidlertid avhjelpe dette.

Slik kan det se ut, når man vil ha alle studenter som har visse feltverdier – det må skilles mellom hel- og deltidsstudenter:

```
public List<Student> finnQBE(int studnr,string studnavn, float prosent)
{
    Student proto = new Student(studnr,studnavn); //posit
    if (prosent > 0)
        proto = new Deltidsstudent(studnr, studnavn, prosent); //admit
    List<Student> studliste=new List<Student>();
    foreach (Student stud in objBase.QueryByExample(proto))
         studliste.Add(stud);
    return studliste;
}
```

QueryByExample returnerer et *IObjectSet*, som må konverteres til noe fornuftig, her en *List*. Hvis jeg vil ha alle studenter som heter "Knut", kaller jeg funksjonen slik:

```
List<Student> studenter = kontroll.finnQBE(0, "Knut", 0);
```

Jeg oppgir standardverdiene 0 for *studnr*, og 0 for prosent og stiller bare krav til *studnavn* som skal være lik "Knut".

For de aller fleste applikasjoner blir QBE for enkelt.

NQ

Med "Native Query" kan jeg skrive spørringen med vertsspråket – her C#. Dermed får jeg full syntakskontroll, typesjekking osv, altså alle fordelene ved å skrive "native code".

Prinsippet går ut på å bruke en "delegate", dvs jeg definerer en referanse til en funksjon med en viss signatur. Funksjonen som det vises til skal være en "predikatfunksjon", dvs en funksjon som returnerer true/false. Her vil da true innebære at objektet skal være med i resultatet. Man kan deklarere en delegate med navn, skape en instans av den osv, men det er unødvendig tungt her. Det er enklere å bruke en anonym delegate.

Jeg bruker da *objBase.Query* og gir den en delegate funksjon som jeg definerer der og da. Siden *Query* returnerer en *IList*, deklarerer jeg en slik liste og tilordner den:

```
public List<Student> finnNQ(float prosent)
{
    IList<Deltidsstudent> Istudliste; //Query returnerer IList
    //Alternativ A - Anonym delegate
    Istudliste = objBase.Query<Deltidsstudent>
        (delegate(Deltidsstudent s)
            { return s.getProsent() >= prosent * .8
                && s.getProsent() <= prosent * 1.2; });
```

En av variantene av *Query* tar et predikat, og det er den jeg bruker. Inne i "spissklammene" finnes uttrykket som evalueres til true/false. Dette vil gi en *IList* med alle deltidsstudenter med prosentverdi som den oppgitte prosenten +/- 20%.

Med Lambda-operatoren

Ved å bruke lambda-operatoren => kan dette skrives enda enklere. Lambda-operatoren kan tolkes som "sendes til" og kan brukes direkte i koden (såkalt "inline"). Med Lambda-operatoren, blir NQ enda enklere:

```
//Alternativ B - Delegate med Lambda-operator
Istudliste = objBase.Query<Deltidsstudent>
    (s => s.prosent >= prosent * .8 && s.prosent <= prosent * 1.2);
```

Her blir alle objektene *s* i *objBase* "sendt til testing" for prosent. Hvis Lambda returnerer true, skal objektet være med i samlingen. Kompilatoren finner selv ut at *s* må være en *Deltidsstudent*.

Lambda-operatoren kan også brukes sammen med en navngitt delegate, her delegaten *dele*:
```
delegate int dele(int i);
dele minDelegate //deklarerer en instans med int-parameter og int retur
…
minDelegate = x => x * x; //definerer instansen
int j = minDelegate(5);  //j = 25
minDelegate = y => y * 5; //definerer instansen igjen og annerledes
j = minDelegate(3); //j = 15
```

Jeg deklarerer altså først en generell funksjon *dele* som representerer en hvilken som helst funksjon med et *int* argument og returverdi. Så deklarerer jeg en instans av den og tilordner instansen en algoritme som kvadrerer heltallet med Lambda-operatoren og bruker den. Deretter tilordner jeg instansen en algoritme som multipliserer med fem og bruker den. Poenget er altså at jeg bruker to forskjellige algoritmer som begge har navnet *minDelegate*. Funksjonen opptrer som en slags variabel som kan tilordnes en algoritme. Det sjekkes at algoritmen bruker riktig datatype som input og returnerer riktig type.

LINQ

For å bruke LINQ må jeg referer til biblioteket Db4objects.Db4o.Linq. For enkelhets skyld, føyer jeg også til
```
using Db4objects.Db4o.Linq;
using Db4objects.Db4o.Query;
```

Med LINQ skriver jeg en spørring som likner på SQL:
```
public List<Student> finnLINQ(string studnavn)
{
    List<Student> studliste = new List<Student>();
    var funnet =
        from Student stud in objBase
        where stud.getStudnavn().ToLower().Contains(studnavn)
        orderby stud.getStudnavn() descending
        select stud;
    foreach (Student stud in funnet)
        studliste.Add(stud);
    return studliste;
}
```
Datatypen *var* er en type som tilpasser seg den aktuelle – en form for generisk, enkel datatype. Kompilatoren finner selv ut hvilken type det er, og tilpasser seg den. Deretter kontrollerer kompilatoren at all bruk av variabelen er av den riktige typen. Akkurat her er det veldig kjekt med *var* fordi LINQ-spørringen returnerer en liste av en intern type.

Legg merke til at spørringen kan *sorteres*. Det er også tillatt å *gruppere* evt. med noe tilsvarende *having*.

Det er fullt mulig å søke blant studenter men returnere noe annet, f.eks.
```
var funnet =
    from Student stud in objBase
    where stud.getStudnavn().ToLower().Contains(studnavn)
    select stud.getStudkurs();
```
Denne vil returnere en liste med kurslister, nemlig *studkurs* for visse studenter.

Kompilatoren vil forsøke å "oversette" LINQ til SODA (se neste punkt) og er derfor vanligvis meget rask å eksekvere.

SODA
Med SODA er tanken at man følger referansene fra node til node i den grafen som er lagret i objektdatabasen. Man starter med en *Query* (av klassen *IQuery*) og alle objekter i objektdatabasen[31]. Med *Descend* følger man en referanse til alle objektene som ligger i spørringen til en ny node. Med *Constraint* innskrenker man søket til å gjelde noen av de nodene man har funnet.

Descend er en operasjon som er knyttet til et *IQuery*-objekt og resultatet er en *IQuery*. Da kan man altså gå videre med en ny *Descend*. *Constrain* er en operasjon knyttet til en *IQuery, men resultatet er en IConstraint* og derfor kan man ikke da gå videre, verken med *Descend* eller *Constraint*. (*Constrain* er følgelig det siste man skriver på en linje.)

Effekten av *Descend* varierer med hva slags nodereferanse man følger:

- ✓ *Descend* til en primitiv variabel.
  Antallet man finner øker ikke, men man får muligheten til å stille krav til variabelen med *Constrain*.
- ✓ *Descend* til en enkelt referanse.
  Antallet øker ikke, men man får muligheten til å *Descend* videre, eller stille krav til det objektet man kommer til med *Constrain*.
- ✓ *Descend* til en struktur.
  Antallet øker. Man får muligheten til å gå videre med *Descend*, eller stille krav med *Constrain*.

I programeksemplet vil jeg finne alle kurs som har deltidsstudenter med prosent mindre enn en oppgitt prosentsats. Jeg starter med å skape en spørring kalt *spørring*. Den inkluderer samtlige objekter i objektdatabasen. Deretter begrenser jeg søket til objekter av typen *Kurs*. Så følger jeg referanse til alle nodene referert til i kursenes liste *kursdeltakere* og begrenser utvalget til dem som er deltidsstudenter. Deretter følger jeg de samme referansene til i *kursdeltakere* og begrenser til bare dem som har mindre prosent enn den oppgitte. Endelig gjennomfører jeg spørringen med *Execute*. Koden ser slik ut:

```
public List<Kurs> finnKursSODA(float maksProsent)
{
    IQuery spørring = objBase.Query();
    spørring.Constrain(typeof(Kurs));
    spørring.Descend("kursdeltakere").Constrain(typeof(Deltidsstudent));
    spørring.Descend("kursdeltakere")
                    .Descend("prosent").Constrain(maksProsent).Smaller();
    IObjectSet funnet = spørring.Execute();
```

Etter dette har jeg fått en mengde objekter (*IObjectSet)*, som jeg legge inn i en liste for retur.

[31] Man kan da straks (som parameter) angi hvilken objektklasse man er interessert i – f.eks. *objBase.Query(typeof(Student))* – men da endres resultatet til et *IObjectSet* som ikke kan brukes videre med *Descend* og *Constraint*. Det er derfor vanlig å bare benytte *objBase.Query()*.

I neste eksempel leter jeg etter deltidsstudenter som har prosent som angitt +/- 20%. Jeg bygger opp tre, forskjellige *constraints*. De knyttes til spørringen som oppgis i kallet. Deretter legger jeg *constraints* med constraintsmetoden *X.And(Y)* – som legger *Y* til *X* med logisk *AND*. Så er det bare å eksekvere.

```
IQuery spørring = objBase.Query();
IConstraint vilkår1 = spørring.Constrain(typeof(Deltidsstudent));
IConstraint vilkår2 = spørring.Descend("prosent")
      .Constrain(prosent * .8).Greater();
IConstraint vilkår3 = spørring.Descend("prosent")
      .Constrain(prosent * 1.2).Smaller();
vilkår1.And(vilkår2).And(vilkår3);
IObjectSet funnet = spørring.Execute();
```

(Vilkårene kunne også vært satt sammen med *And* direkte, men jeg synes dette er enklere å holde orden på.)

Oppsummering om søking med Db4O

Alle søkemetodene har fordelen av full syntaks og typekontroll, men SODA oppgir feltnavn og de er strenger, så det kan kompilatoren ikke sjekke.

Etter mitt syn er QBE ganske primitiv. Jeg misliker sterkt at den krever en argumentfri konstruktør. Dessuten er det kun mulig å søke på likhet med oppgitte feltverdier og det er kun logisk *and* mellom dem. Når det da i tillegg er umulig å søke etter objekter med default initialverdier (0 for tall, null for strenger og andre objektreferanser og false for boolske variable), blir det hele for enkelt. Endelig vil jeg nevne at alle objekter i objektdatabasen må sammenliknes med prototypen, og det er meget tregt. QBE kan følgelig kanskje brukes i kurs for brukere som skal delta i et prosjekt, for å gi dem en følelse av hva som foregår.

NQ er jeg selv "glad i". Db4objects anbefaler den for språk som ikke støtter LINQ. Den store fordelen, synes jeg, er at man bruker vertsspråket som er godt kjent for programmereren. Det er således lite nytt å lære seg, og det er meget fleksibelt. Db4o vil forsøke å utnytte indekser og referanser så langt det går, slik at færrest mulig objekter faktisk må hentes fra objektdatabasen. Ulempen er at hvis db4o ikke klarer det, så blir spørringen treg.

Db4objects anbefaler selv LINQ i .NET-miljøer. LINQ er relativt lett å skrive, den er fleksibel og den interpreteres til SODA om mulig. Hvis det ikke er mulig kan LINQ bli tung. LINQ er grei å skrive og vil virke kjent for dem som kjenner SQL for relasjonsdatabaser. Det er en god innfallsport, f.eks. i kurs for databasefolk.

SODA er ikke helt enkel å skrive, og den krever mer kjennskap til den faktiske grafen som er lagret i objektdatabasen. Imidlertid er den raskere enn de andre metodene, fordi den i stor grad følger referansene i databasen. Det er bare sjelden bruk for å hente objektene fra objektdatabasen. Selv har jeg problemer med den når jeg ikke får forventet resultat og ikke forstår hva som er galt. Det er ikke så mye syntaks, men logisk er den noe vanskelig å forstå. Gjør man en feil her, vil spørringen simpelthen ikke returnere noe. SODA gir ikke mye feilmeldinger når den eksekveres, og det er ikke mulig å "følge" eksekveringen når spørringen utføres (da navigerer db4ob omkring i objektdatabasen og det synes ikke i programmet mitt). Jeg vil allikevel foretrekke SODA i applikasjoner der eksekveringshastigheten er kritisk, f.eks. når den skal kjøres på en server med stor pågang fra klienter eller der objektdatabasen er stor.

Kompilatoren vil uansett forsøke å gjøre om NQ og LINQ til SODA-spørringer og derved oppnå hastigheten ved SODA. Det er imidlertid ikke sikkert at den klarer det, og da kan det være aktuelt å skrive SODA-spørringen selv, direkte. Annen bruk av SODA anbefaler db4objects ikke.

Konklusjon

Valg av spørremetode er spørsmål om smak og situasjon:

- ✓ QBE for enkle brukerkurs – kan vise hva objektdatabase går ut på.
- ✓ NQ for dem som liker å programmere – gir god kontroll. Anbefales av db4objects for miljøer uten LINQ. Forsøkes oversatt til SODA av kompilatoren.
- ✓ LINQ for dem som kan SQL og ikke vil lære for mye nytt. Anbefalt av db4objects for .NET-miljøer. Kompilatoren forsøker å oversette til SODA.
- ✓ SODA når fart er kritisk, ellers litt vanskelig logikk og ikke anbefalt.

G: Persistens ved hjelp av logging (Prevayler)

Generelt

Dette er jo så enkelt at det nesten virker for godt til å være sant. Det er imidlertid rapportert at dette faktisk virker, og det er meget raskt – det er svært lite overhead med loggingen. Ved oppstart kan det ta noe tid, men det er vanligvis ikke kritisk. Det er uansett snakk om sekunder, selv ved store antall objekter.

Jeg har laget en testbenk på min bærbare PC. Det tok typisk fra ett til tre hundredels sekunder å starte programmet uten noen objekter. Jeg genererte så 20 000 objekter som ble lagt i en liste. Det tok vel 5 minutter og loggfilen ble på vel 10 Mb. Deretter stanset jeg programmet og startet det igjen. Da tok det bare ca tre hundredels sekund å gjenopprette alle objektene i RAM – tiden det tar å gjenopprette objektene er praktisk talt ikke merkbar. En "snapshot" ble laget og da ble filen bare 0,4 Mb (i dette tilfellet er kopien av meldingene i loggfilen betydelig større enn objektene i "snapshot"-filen). Det var ikke merkbar forskjell på tiden det tok å gjenopprette "snapshot". Man må anta at en større server ville gjøre alt raskere – de nevnte tidene gjelder min bærbare PC, men er ganske imponerende.

Tankegangen bak

Istedenfor å lagre objektene i en objekt- eller annen database, kan man logg alt som skjer. Ved hjelp av denne loggen, kan man senere gjenopprette internminnet.

Alle tilstandsendringer skjer altså som resultat av meldinger til kontrollobjektet. (Det er nettopp slik jeg har lagt opp mitt demonstrasjonsprogram.) Disse meldingene logges. Når systemet tas opp igjen senere, sendes de lagrede meldingene pånytt til kontrollobjektet. Siden resultatet av alle meldinger er deterministisk, vil tilstandsendringene bli de samme. Når hele loggen er "kjørt" slik, er systemet tilbake i den tilstand det var da loggingen ble avsluttet. Da kan systemet igjen behandle meldinger fra klienter, og systemet er i drift igjen. De nye meldingene må også logges.

Meldinger gir ofte respons. Det kan være data som meldingen ba om, eller respons på en tilstandsendring som det er bedt om – et vanlig svar eller et feilobjekt. Når systemet er i drift, returneres slike til klienten. Når systemet er under gjenopprettelse, er det et spesielt objekt som sender meldingene fra loggen til kontrollobjektet. All respons vil da returneres dit, og dette objektet ser helt bort fra alle slike – også feil.

Meldinger som bare henter data uten å endre tilstand, kan gjerne logges de også. Da er det unødvendig å skille mellom de meldingene som endrer tilstand og alle andre. Prisen er at loggen blir større, og det vil ta lengre tid når systemet skal gjenopprettes vha loggen.

"Snapshot"

Etter at systemet har vært i drift en stund, vil loggen bli stor. Den vil da innehold mange tilstandsendringer som senere er gjort om. Det har da ingen hensikt å gjenta dem når systemet gjenopprettes. Det kan da tas en "snapshot". Da serialiseres samtlige objekter i RAM til en egen fil og loggingen fortsetter. Neste gang systemet skal gjenopprettes, utføres først deserialisering av alle objektene fra "snapshot". Deretter gjentas bare logger som er laget *etter* "snapshot". Logger fra tiden før "snapshot" ble tatt, er altså da overflødige. Det vil ta litt tid å lage en "snapshot" når det er mange objekter. Den tiden får man fort igjen ved neste gjenoppretting. Programmereren kan bestemme at skal lages "snapshot" ved faste tidspunkt, etter et visst antall meldinger, hver gang programmet tas ned, på brukerens anmodning eller en hvilken som helst passende strategi. Ved systemfeil vil det ikke bli laget "snapshot" men loggene finnes jo allikevel, så systemet kan gjenopprettes. Det er eneste man oppnår med en "snapshot" er å spare tid ved neste gjenoppretting.

Etter en "snapshot" må det startes en ny logg for alle meldinger som ankommer etter "snapshot" ble tatt. Dette likner på *redo* i en database, som starter med seneste backup og kjører alle (kommitterte) transaksjoner fra loggen etter backup.

Som det fremgår er Prevayler *ikke* en objektdatabase, men en *loggfil*. Loggfilen er rask, fordi den bare skal legge til sekvensielt (append). Farten kan økes ytterligere ved å sette av stor og kontinuerlig plass til loggfilen på forhånd.

Med et slikt system, er commit point i det øyeblikk meldingen er logget til disk. Da er endringen blitt konsistent.

Siden alle objektene ligger i (én) RAM og kontrollobjektet styrer alle henvendelser, er det ingen problemer med transaksjonsfletting – alt blir nødvendigvis ACID.

Prevayler

Et slikt logg-system er laget av brasilianeren Klaus Wüstefeld. Klaus kalte det *Prevayler* og prinsippet for *Prevalence*[32]. Klaus laget et slikt system i form av klasser for Java. Siden alt skjer med POJO ("Plain Old Java Objects") kan alt sjekkes av kompilatoren med typesjekking og det hele.

Siden han laget *Prevayler* har Klaus sloss med besserwissere som forsøker å skyte ned idéen. Selv synes jeg at både Klaus (jeg har truffet ham) og idéen er kul. Klaus er veldig langt fra å være A4 konform og har en frisk tone som mange lar seg erte opp av. Han er imidlertid en hyggelig kar privat og det virker som han har det gøy.

Han jobber nå faktisk med db4o for db4Objects (under Versant)!

Senere har fri programvare prosjektet Bamboo Prevalence laget tilsvarende for .NET, som jeg skal demonstrere her.

Man trenger biblioteket Bamboo.Prevalence[33]. Det må refereres til dette biblioteket, og jeg henviser gjerne også til det med "using":

```
using Bamboo.Prevalence;  //Bamboo - krever referanse
using Bamboo.Prevalence.Attributes; //Bamboo - inkludert i referansen Prevalence
```

Programeksempel: Persistens ved hjelp av logging (Prevayler)

Prevayler og *Bamboo Prevalence* stiller følgende krav:

1. Alle tilstandsendringer går gjennom ett kontrollobjekt
2. Alle objektene ligger i RAM til enhver tid ("RAM-based system")
3. Alle objektene er deterministiske, dvs. at to like meldinger – inkludert konstruktøren – som har de samme argumentene, alltid fører til samme resultat. Da kan meldingen ikke medføre
    a. Randomiserte prosesser slik at resultatet er avhengig av tilfeldigheter. Da vil gjentatt bruk av meldingen ikke alltid gi samme resultat som forrige gang. (Det finnes kanskje applikasjoner der dette ikke spiller noen rolle, f.eks. simuleringer. Men også der vil statiske mål da bli annerledes.)
    b. Referanse til systemklokken, f.eks. en timestamp for opprettelse. Når loggen

[32] *Prevalence* = hyppighet av en sykdom i en befolkning. Klaus mener IT-folks bastante tro på relasjonsdatabaser er som en kreftsykdom som de ikke vet at de har. Prevayler er kuren som kan gjøre dem friske.
[33] Finnes på http://sourceforge.net/projects/bbooprevalence/files/

kjøres for gjenoppretting har systemet en annen tid og resultatet blir annerledes. (Dette kan lett løses ved at konstruktøren tar klokkeslettet som argument. Det vil da bli en del av loggen og det gjenopprettede objektet blir nøyaktig likt det opprinnelige.)

4. Hvis det skal tas "snapshot" kreves det også at
   a. Alle objekter er serialiserbare
   b. Alle objekter har en argumentfri konstruktør

Entitetsklassene

Her kreves kun de endringer som er nødvendig for "snapshot", dvs. de må være serialiserbare og ha en argumentfri konstruktør.

Skjemaklassen

Den eneste endringen som er aktuell her, er evt. å lage en egen knapp så brukeren selv kan bestemme når det skal tas "snapshot".

Det er ikke nødvendig å kalle henting og lagring, det er automatisk med *Prevayler*.

Kontrollklassen

Kontrollklassen gis arv fra *MarshalByRefObject*[34] (en klasse i .NET):

```
[Serializable]
public class Kontroll : MarshalByRefObject //MarshalByRef kreves av Bamboo
```

MarshallByRefObject gjør det mulig å skape proxy-objekter av kontrollklassen.

Det deklareres to *static* objekter:

```
static private PrevalenceEngine minEngine; //Bamboo
public static readonly Kontroll kontroll = new Kontroll();
```

Når kontrollobjektet skapes, må *Prevayler* aktiveres:

```
static Kontroll() //så ingen andre kan instansiere Kontroll
{
    //Snapshot & logger leses fra den lokale katalogen "bamboodata"
    string bambookatalog = Environment.CurrentDirectory + "\\bamboodata";
    minEngine = Bamboo.Prevalence.PrevalenceActivator
        .CreateTransparentEngine(typeof(Kontroll), bambookatalog);
    kontroll = (Kontroll)minEngine.PrevalentSystem;
}
```

Dette gjør det mulig for objektet *minEngine* som tilhører en anne applikasjon, å få tilgang til *kontroll*-objektet vha av et proxy-objekt (som sies å være *transparent*). Det er da *minEngine* som logger og gjenoppretter tilstand.

Når det ovenstående er gjort, trenger man ikke lenger å tenke på persistens i det hele tatt. Alle meldinger til kontrollobjektet logges og systemets tilstand gjenopprettes automatisk ved neste oppstart.

[34] MarshallByRef er en teknikk som brukes når et objekt fra én applikasjon (her: Prevayler) vil ha tilgang til et objekt fra en annen applikasjon (her: vårt program). Man sender ikke objektet selv eller en referanse til objektet til en annen applikasjon, for der er objektet ubrukelig. Isteden sendes referanse til en "proxy". Proxy-objektet inneholder det som er nødvendig for aksess til det egentlige objektet. Grensesnittet for proxy-objektet er gjerne det samme som for det egentlige objektet (og det er slik her), men sørger for at meldinger som ankommer videresendes til det egentlige objektet. Samtidig kan proxy-objektet gjøre andre oppgaver, som f.eks. her å logge meldingene og gjenskape forrige tilstand ved oppstart.

Oppsummering om persistens av objekter

Ovenfor har jeg forklart hvorfor det er nødvendig å sikre persistens av objektene i et objektorientert system mellom sesjoner (kjøringer av det objektorienterte systemet). Jeg har vist hvordan persistens av objekter kan oppnås på mange forskjellige måter. Jeg har forklart og vist eksempler på seks helt forskjellige måter, noen med varianter. Alle eksemplene er basert på det samme, tenkte systemet, slik at det skal være mulig å sammenlikne metodene direkte.

Nedenfor sammenlikner jeg metodene utfra noen aspekter som er av betydning når man skal velge persistensmetode. Jeg mener at primært er følgende forhold viktige:

1. Lagringen kan medføre én eller flere filer. Det kan være en fordel ved sikkerhetskopiering, flytting og andre administrative operasjoner at dataene er i bare én fil.
2. Lagringen kan være slik at dataene enkelt kan importeres til andre systemer, f.eks. for ytterligere analyse.
3. Lagringen kan være slik at det er mulig å få tilgang til dataene (objektenes egenskapsverdier) på andre måter enn gjennom det objektorienterte systemet. F.eks. vil lagring i en kommalimitert, flat fil være mulig å redigere uten å bruke det objektorienterte systemet. Det kan være en fordel ved at det er enkelt å gjøre mindre rettelser, men medfører fare for feilendringer.
4. Lagringen kan stille krav til det objektorienterte systemet, f.eks. vil en database kreve at det finnes en verdi som identifiserer hvert objekt (en "primærnøkkel"). I objektorientert programmering brukes ikke en slik ID – der er det objektets adresse i RAM som er objektets ID. Andre krav kan være at alle objektene som skal lagres er referert fra én samling (collection), at det ikke finnes sirkelreferanser (rekursjon). Videre kan jeg nevne krav om at visse deler av objektene gjøres "public" utover det som programmerere flest vil ønske og at det må finnes bestemte former for konstruktør. Alle slike krav binder programmereren til løsninger som ikke er "naturlige" i objektorientert programmering.
5. Lagringen kan kreve at objektene må konverteres eller mappes fra objektorientert form til den formen lagringen krever. Tilsvarende må da den lagrede strukturen konverteres tilbake til objekter ved henting fra lageret. Dette kan være mer eller mindre tungvint. I noen tilfeller finnes det en ferdigdefinert klasse som ordner konverteringen og mappingen, i andre tilfeller må programmereren selv skrive denne koden. Det gir ekstra arbeid og ikke minst risiko for feil. Noen lagringssystemer søker riktignok å avhjelpe slike problemer med automasjon, men også det har sine ulemper.
6. Lagringen kan kreve at alle objektene lagres og hentes på en gang. Da er det altså ikke enkelt å lagre endring av ett, enkelt objekt. Det kan fort bli tidkrevende og krever at alle objektene får plass i RAM samtidig.
7. Hvis det er mulig å hente/oppdatere ett og ett objekt, er det av betydning hva lagringssystemet tilbyr av søkemuligheter. Dataene vil vanligvis legges på en tjener, slik at de er tilgjengelige for flere (flerbruker- kontra enbrukersystemer). Da kan det være en fordel om det er gode muligheter for å la tjeneren foreta søking og bare returnere én, eller noe få data. Alternativet er å hente store mengder data til klienten, som så selv besørger søkingen. Tjenere er gjerne større, kraftigere og følgelig raskere til slik søking og man sparer mye kommunikasjon som også kan være forsinkende.
8. Lagringen kan være rask eller tidkrevende. Generelt er all lagring på platelager en flaskehals i programmer, da alle operasjoner i RAM er svært mye raskere.

Det er disse åtte punktene jeg vil ta opp for hvert lagringssystem for seg.

Jeg vil også nevne at lagringen kan være slik at det ikke er mulig for flere å benytte systemet samtidig. Dette kan vanligvis løses ved å legge deler av systemet på en tjener. På tjeneren vil man da legge de delene som håndterer persistensen. Jeg anser følgelig ikke dette som noe stort problem og ser bort fra det nedenfor.

Konklusjon for flat, kommalimitert (csv) fil

Med en slik løsning lagres objektenes egenskaper som data i en flat fil med skilletegn (ofte komma) mellom hver verdi og linjeskift mellom objektene. Dette er en ren tekstfil.

1. **Ulempe:** Det er vanskelig å holde seg til én fil – det blir gjerne flere filer.
2. **Fordel:** Kommalimiterte filer kan importeres til mange standardsystemer.
3. **Fordel:** Det er mulig å åpne filen med en enkel editor. Derved kan man både sjekke og endre data enkelt. Dette krever riktignok forståelse for filens struktur og medfører fare for feilendinger.
4. **Fordel:** Kommalimitert fil stiller ingen krev til det objektorienterte systemet.
5. **Ulempe:** Det kreves egen mapping og konvertering til/fra filen(e). Dette er relativt enkelt for hvert objekt, men tyngre og med risikofylt for sammenhenger mellom dem.
6. **Ulempe:** Det er vanskelig å endre eller slette ett, enkelt objekt som er lagret. Derimot kan det være enkelt å lagre ett nytt objekt.
7. **Ulempe:** Det finnes ingen søkemuligheter for lagringen.
8. **Ulempe:** Fordi alle objektene må lagres samtidig (med unntak for nye objekter) vil de fleste systemer får lagringen som en flaskehals.

Hovedkonklusjonen er da at kommalimitert fil er gunstig ved at den er ren tekst som kan inspiseres og editeres i enkle editorer. Den har et format som mange andre systemer kan importere for ytterligere analyse. Det er enkelt å ta sikkerhetskopi o.l. Det stiller ingen krav til det objektorienterte systemet. For øvrig er det lite som taler for å benytte slik lagring. Dessuten er det ikke bra at alle objekter må lagres/hentes på en gang.

Konklusjon for serialiserte objekter (binær form)

Objektene lagres da i en binær form. Lagringen gjennomføres med klasser som er ferdig definert, uten noen mapping.

1. **Fordel:** Lagring skjer vanligvis i én fil.
2. **Ulempe:** Filformatet er slik at det ikke er mange – om noen – andre systemer som kan importere dataene.
3. **Ulempe:** Det er ikke mulig å få tilgang til dataene på andre måter enn gjennom et objektorientert program. Fordelen er da at de heller ikke kan feilredigeres.
4. **Ulempe:** Binær lagring kan bare skje av ett, eneste objekt. I det objektorienterte programmet må da alle objektene refereres fra en samling og det er denne samlingen som lagres. Videre må klassene merkes *[Serializable]*, men det medfører vanligvis ikke noe problem. Det er en fordel at alle referanser mellom objektene lagres samtidig.
5. **Fordel:** Lagringen krever ingen konvertering eller mapping. Det finnes en ferdig klasse (*BinaryFormatter*) som gjør hele arbeidet, så det er svært enkelt å programmere.
6. **Ulempe:** Alle objektene må lagres og hentes på en gang.
7. **Ulempe:** Det er ingen søkemuligheter i den binære filen.
8. **Ulempe:** Lagringen er tidkrevende.

Hovedinntrykket er at binær lagring først og fremst er enkelt for programmereren. Det er vanskelig å finne andre fordeler av betydning. Dessuten er det ikke bra at alle objekter må lagres/hentes på en gang.

Konklusjon for XML-fil

Når objektene lagres i en XML-fil, benyttes "tagger" til å angi klasser og egenskaper. Filen er en ren tekstfil.

1. **Fordel:** Lagringen medfører vanligvis bare én fil.
2. **Fordel:** XML-fil er et standardisert format – mange andre systemer kan importere slike filer.
3. **Fordel:** Filen er lesbar for mennesker og kan redigeres av dem i en vanlig editor eller en spesialisert XML-editor. Det er dog en fare for feilredigeringer.
4. **Ulempe:** Alle egenskapene må være *Public*, så all informasjonsskjuling ødelegges. Videre må alle objekter refereres fra én samling (collection). I sin enkleste form (*XML-serializer*) godtas heller ikke sirkelreferanser mellom objekter, men en mer kompleks klasse (*DataContractSerializer*) aksepterer det.
5. **Fordel:** Det kreves ingen mapping og konverteringen er automatisk, når man bruker de ferdigdefinerte klassene som tilbys.
6. **Ulempe:** Lagringen krever at alle objektene lagres og hentes på en gang.
7. **Ulempe:** Det er ingen søkemuligheter.
8. **Ulempe:** Lagringen er tidkrevende – alle objektene lagres og hentes på en gang.

Hovedkonklusjonen er da at XML-fil er gunstig ved at den er ren tekst som kan inspiseres og editeres i enkle editorer og i spesialiserte XML-editorer. Den har et format som mange andre systemer kan importere (men ikke de samme som kan importere kommalimiterte filer). Det er enkelt å ta sikkerhetskopi o.l. Videre er det enkelt for programmereren da ferdigdefinerte klasser håndterer all konvertering.

Det er en alvorlig innvending at de klassene som skal håndtere konvertering og lagring/henting, krever at alle egenskaper er *public*. Det er jo vanlig at objektorientert programmering anvender informasjonsskjuling maksimalt. Dessuten er det ikke bra at alle objekter må lagres/hentes på en gang.

Konklusjon for relasjonsdatabase

Mange virksomheter benytter i dag relasjonsdatabaser til sine data. De benytter slik database også til å lagre objekter fra objektorienterte systemer. Mange av de mest brukte programmeringsspråkene som benyttes i dag er objektorienterte (Java, php, C++, C#, perl – et unntak er C). Samtidig er relasjonsdatabaser helt klart mest brukt. Det tyder på at dette er den vanligste måten å lagre objekter på.

1. **Fordel:** Lagringen skjer riktignok ikke i én fil, men i et databasehåndteringssystem. Der er det innebygget mange muligheter for sikkerhetskopiering osv.
2. **Ulempe:** Det er vanskelig å importere data fra en database til andre systemer, f.eks. et regneark. På den annen side, kan databaser enkelt aggregeres til ledelsesinformasjonssystemer (LIS/MIS) og mange programmer kan aksessere databaser.
3. **Fordel:** Det finnes mange verktøy som gir tilgang til dataene uten å gå igjennom det objektorienterte systemet.

4. **Ulempe:** Relasjonsdatabaser stiller krav om primærnøkkel (og fremmednøkler). På den annen side har ofte brukerne sine identifiserende egenskaper uansett, så ulempen er ikke stor.
5. **Ulempe:** Lagring i database krever både mapping og spesialskrevne rutiner for konvertering til/fra objekter. Det finnes riktignok automatiserte systemer som skal mappe og strukturere relasjonsdatabasen, men også de har sine ulemper – ikke minst med nye versjoner av klassene.
6. **Fordel:** Relasjonsdatabaser kan lagre enkeltobjekter for seg. Det er ikke nødvendig å ha alle objektene samtidig i RAM.
7. **Fordel:** Det er betydelige søkemuligheter i relasjonsdatabaser med egne verktøy for formålet. Det er fullt mulig og vanlig å overlate søkingen til tjeneren der databasen ligger.
8. **Fordel:** Lagringen er relativt rask for ett og ett objekt.

Jeg vil også nevne at relasjonsdatabaser gjør det enkelt å knytte det objektorienterte systemet til andre av virksomhetens systemer. Ja, de vil vanligvis benytte samme database for dem alle. Videre er relasjonsdatabaser en godt kjent teknologi og alle nyutdannede kjenner dem godt.

Hovedkonklusjonen er at det store problemet er mapping og konvertering, med mye arbeid og store feilmuligheter. Det kan være en ulempe at det kreves identifiserende egenskap i alle klasser. For øvrig er relasjonsdatabaser en velegnet lagringsmetode.

Konklusjon for objektdatabase

Objektdatabaser lagrer objektene på samme form. De lagres som objekter og hentes som objekter. Referanser mellom objekter lagres tilsvarende. Objektorienterte databaser kan beskrives som at de lagrer hele den objektorienterte grafen, med alle noder og referanser.

1. **Fordel:** Flere objektdatabaser lagrer alle objekter i én fil.
2. **Ulempe:** Lagringen skjer i et format som bare er kjent for databasesystemet selv. Det er ikke mulig å importere dataene til andre systemer.
3. **Ulempe:** Databasen kan kun leses med et objektorientert system. Det lagres i en form som hverken er lesbar eller redigerbar for mennesker. Imidlertid finnes det måter å lage ad hoc spørringer på.
4. **Fordel:** Lagring i en objektorientert database, stiller ingen krav til det objektorienterte systemet. Programmereren kan programmere objektorientert slik han ønsker.
5. **Fordel:** Lagringen krever hverken mapping eller konvertering. Den konverteringen som evt. skjer er helt automatisk.
6. **Fordel:** Det er vanligvis ikke noe krav om at alle objektene skal lagres/hentes samtidig. Det er enkelt å lagre/hente ett og ett objekt, evt. med subobjekter.
7. **Ulempe:** Det tilbys svært gode søkemuligheter – på mange måter - så tjeneren kan foreta søkingen, og den er svært rask fordi den utnytter grafen under søkingen.
8. **Fordel:** Lagringen er meget rask.

Hovedkonklusjonen er at objektdatabaser virker svært godt tilpasset objektorienterte programmer. De er svært enkle å bruke for programmereren. Det kan sies å være en ulempe at dataene bare er tilgjengelig gjennom objektorienterte systemer – import eller manuell inspeksjon/redigering er ikke mulig. Da er jo riktignok risikoen for feilredigering også eliminert.

Når slike databaser er lite brukt (det snakkes om en markedsandel på rundt to prosent), tror jeg det kan skyldes at slike databaser i liten grad undervises ved universiteter/høyskoler, samt at relasjonsdatabaser har fått en så solid markedsandel at det kan være vanskelig å skifte til noe annet.

Konklusjon for logging

Ved logging lagres ikke objektene i det hele tatt. I stedet logges alle endringer av systemets objekter. Ve å gjenta alle endringen fra "tom" RAM, gjenskapes systemet.

1. **Fordel:** Lagringen gjøres i én/flere loggfil(er) – alle i samme mappe.
2. **Ulempe:** Loggfilene kan ikke importeres til andre systemer. De er laget for å gjenoppbygge RAM og objektene lagres egentlig ikke – bare de handlinger som skapte og vedlikeholdt dem.
3. **Ulempe:** Det er ingen mulighet til å få tilgang til objektene på annen måte.
4. **Ulempe:** Det stilles krav om at alle objektene er deterministiske (ellers kan de ikke gjenskapes gjennom loggingen) og det inkluderer forbud mot referanser til systemklokken. Begge deler ordnes imidlertid ganske enkelt. Videre kreves det at alle tilstandsendringer gjøres ved kall til bare ett objekt. Det er dette objektet som logges. Endelig kan jeg nevne at visse operasjoner kan kreve at klassene er *Serializable* og har argumentfri konstruktør, uten at det gir særlige problemer. Det er allikevel slik at ingen av kravene – muligens med unntak av kravet om argumentfri konstruktør – strider mot prinsippene i objektorientert programmering.
5. **Fordel:** Det kreves ingen mapping og ingen konvertering. Det er ikke bare automatisk men unødvendig.
6. **Fordel:** Det er ingen lagring av objekter og dermed heller ikke noe krav at alle lagres på én gang. Isteden er det et krav at alle objektene ligger samtidig i RAM og det kan skape problemer.
7. **Ulempe:** Det finnes ingen søkemuligheter i systemet. Programmereren må skrive kode for all søking selv i RAM. TIl gjengjeld er det alltid meget raskt.
8. **Fordel:** Det lagres bare én og én begivenhet etter hvert som de skjer, og hver av dem kan legges til bakerst i en loggfil Det er meget raskt.

Hovedkonklusjonen er at logging er en kreativ måte å løse persistensproblemet på. Det er meget raskt og krever svært lite av det objektorienterte systemet. Det kan imidlertid være en alvorlig ulempe at all behandling av dataene må skje gjennom det objektorienterte programmet som håndterer objektene.

Sammenlikning av persistensmetodene

Som alltid vil valget mellom persistensmetodene avgjøres av bruken. Hvis det er viktig at objektene lagres i en form som er manuelt redigerbar og kan importeres til andre systemer, så peker XML-fil seg ut som sentral. Da må riktignok alle objektene refereres fra én samling og alle egenskaper må være *public*. Hvis dataene skal importeres til programmer som ikke aksepterer XML, er kommalimitert fil en mulighet. Da må man imidlertid legge betydelig mer arbeid i mapping og konvertering men det stilles ellers ingen krav til programmet. Begge disse har den ulempen at alle objekter må lagres/hentes på en gang.

Hvis det sentrale bare er å sikre persistens – og særlig hvis det gjelder et enbrukersystem – så peker binær lagring seg ut som svært enkel og med liten risiko for feil. Imidlertid krever slik lagring at samtlige objekter samles i én samling (collection).

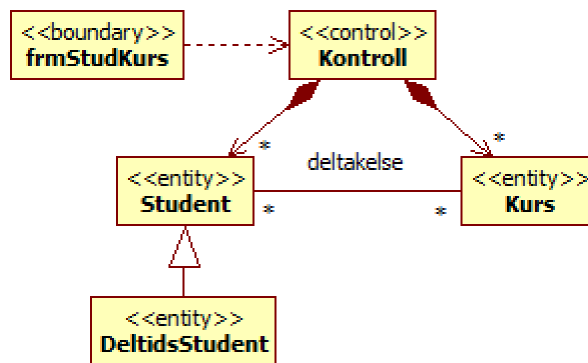
Relasjonsdatabaser bruker en godt kjent teknologi og gir god kobling til andre systemer som bruker databasen, Det er gode søkemuligheter, god sikkerhet og mange andre fordeler. Ulempen er at det medfører betydelig og vanskelig mapping og at konvertering må spesialprogrammeres både til og fra objektformen. Relasjonsteknologien er helt annerledes enn den objektorienterte, hvilket gir den såkalte "impedance mismatch".

I de fleste tilfeller vil en objektdatabase gi meget god tilpasning til det objektorienterte systemet. Det er ingen mapping og ingen konvertering, og det er meget raskt. Objekter kan hentes/lagres ett og ett ved behov og det er meget gode søkemuligheter i objektdatabasen. Det stilles overhodet ingen krav til det objektorienterte systemet, hvilket jeg anser som en stor fordel. Objektorienterte databaser er svært enkle å implementere. Ulempen kan sies å være at det er vanskeligere å koble et slikt system til eksisterende relasjonsbaserte systemer, da det kreves et objektorientert program for å lese dataene.

Logging fremstår først og fremst som en revolusjonerende idé, og den utpeker seg som særlig rask. Den stiller noen enkle krav til det objektorienterte systemet, men er enkel å implementere.

Valget vil da avhenge totalt av bruken, men hvis man ser bort fra enkle enbrukersystemer, mener jeg relasjonsdatabaser og objektdatabaser peker seg ut. Av disse er objektdatabase klar vinner hvis man ikke må ha dataene i en relasjonsdatabase av helt andre grunner. Det vil koste lite – f.eks. en dag eller to med kurs – for å beherske objektdatabaser, så manglende kompetanse er ikke noe godt argument mot dem. Som programmerer har jeg "tapt mitt hjerte" til objektdatabasene.

Vedlegg A – Programeksemplet uten persistens



Modellen for eksemplet *(gjentatt og forenklet – komplett graf side 7)*

Studentkurs - grenseklassen

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using StudentKursLibA;

namespace StudentKurs
{
    public partial class frmStudentKurs : Form
    {
        private Kontroll kontrollAlias
            = Kontroll.kontroll; //alias gjør nedenstående skriving enklere

        public frmStudentKurs()
        {
            InitializeComponent();
        }
        private void frmStudentKurs_Load(object sender, EventArgs e)
        {
            try
            {
                ryddSkjema();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
        private void visStudenter()
        {
            try
            {
                lstStudenter.Items.Clear();
                foreach (Student s in kontrollAlias.getStudenter())
                    lstStudenter.Items.Add(s);
                butSlettStudent.Enabled = false;
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
        private void visKurser()
        {
```

```
try
{
    lstKurser.Items.Clear();
    foreach (Kurs k in kontrollAlias.getKurser())
        lstKurser.Items.Add(k);
    butSlettKurs.Enabled = false;
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
}
private void butSlettStudent_Click(object sender, EventArgs e)
{
try
{
    int studnr = ((Student)lstStudenter.SelectedItem).getStudnr();
    Student slettStudent = kontrollAlias.findStudenter(studnr);
    kontrollAlias.delStudenter(slettStudent);
    ryddSkjema();
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
}
private void butSlettKurs_Click(object sender, EventArgs e)
{
try
{
    Kurs valgtKurs = (Kurs)lstKurser.SelectedItem;
    kontrollAlias.delKurser(valgtKurs);
    ryddSkjema();
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
}
private void ryddSkjema()
{
try
{
    visKurser();
    visStudenter();

    txtNyttStudnr.Clear();
    txtNyttStudnavn.Clear();
    chkNyDeltid.Checked = false;
    txtNyProsent.Clear();
    txtEndretStudnr.Clear();
    txtEndretStudnavn.Clear();
    chkEndretDeltid.Checked = false;
    txtEndretProsent.Clear();
    txtNyKursID.Clear();
    txtNyttKursnavn.Clear();
    txtEndretKursID.Clear();
    txtEndretKursnavn.Clear();
    txtStudnrKobling.Clear();
    txtKursIDKobling.Clear();
}
catch (Exception ex)
```

```
        {
            MessageBox.Show(ex.Message);
        }
    }
    private void lstStudenter_SelectedIndexChanged(object sender, EventArgs e)
    {
        try
        {
            butSlettStudent.Enabled = true;
            Object valgt = lstStudenter.SelectedItem;
            txtEndretStudnr.Text = ((Student)valgt).getStudnr().ToString();
            txtEndretStudnavn.Text = ((Student)valgt).getStudnavn();
            if (valgt.GetType() == typeof(Deltidsstudent))
            {
                txtEndretProsent.Text = ((Deltidsstudent)valgt).getProsent().ToString();
                chkEndretDeltid.Checked = true;
            }
            else
            {
                txtEndretProsent.Text = "";
                chkEndretDeltid.Checked = false;
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
    private void lstKurser_SelectedIndexChanged(object sender, EventArgs e)
    {
        try
        {
            butSlettKurs.Enabled = true;
            Kurs valgt = (Kurs)lstKurser.SelectedItem;
            txtEndretKursID.Text = valgt.getKursID();
            txtEndretKursnavn.Text = valgt.getKursnavn();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
    private void butRegStudent_Click(object sender, EventArgs e)
    {
        int studnr=299;
        float prosent=0;
        Student nyStudent;
        try
        {
            //Kontroll av verdiene skjer i studentklassen
            int.TryParse(txtNyttStudnr.Text, out studnr);
            float.TryParse(txtNyProsent.Text, out prosent);
            if (chkNyDeltid.Checked)
                nyStudent = new Deltidsstudent(studnr, txtNyttStudnavn.Text, prosent);
            else
                nyStudent = new Student(studnr, txtNyttStudnavn.Text);
            kontrollAlias.addStudenter(nyStudent);
            ryddSkjema();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
```

```
        }
    }
    private void butRegKurs_Click(object sender, EventArgs e)
    {
        try
        {
            //Kontroll av ID og connString gjøres av kurs-klassen
            Kurs kurs = new Kurs(txtNyKursID.Text, txtNyttKursnavn.Text);
            kontrollAlias.addKurser(kurs);
            ryddSkjema();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
    private void butRegKobling_Click(object sender, EventArgs e)
    {
        int studnr;
        try
        {
            int.TryParse(txtStudnrKobling.Text, out studnr);
            kontrollAlias.meldinn(studnr, txtKursIDKobling.Text);
            ryddSkjema();
        }
        catch (Exception Exception)
        {
            MessageBox.Show(Exception.Message);
        }
    }
    private void butSlettKobling_Click(object sender, EventArgs e)
    {
        int studnr;
        try
        {
            int.TryParse(txtStudnrKobling.Text, out studnr);
            kontrollAlias.meldut(studnr, txtKursIDKobling.Text);
            ryddSkjema();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }
    private void butEndreStudent_Click(object sender, EventArgs e)
    {
        try
        {
            int studnr;
            float prosent;
            int.TryParse(txtEndretStudnr.Text, out studnr);
            float.TryParse(txtEndretProsent.Text, out prosent);
            if (chkEndretDeltid.Checked)
                kontrollAlias.endreStudent(studnr, txtEndretStudnavn.Text, true,
prosent);
            else
                kontrollAlias.endreStudent(studnr, txtEndretStudnavn.Text, false);
            ryddSkjema();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
```

```
            }
        }
        private void butEndreKurs_Click(object sender, EventArgs e)
        {
            try
            {
                kontrollAlias.endreKurs(txtEndretKursID.Text, txtEndretKursnavn.Text);
                ryddSkjema();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
        private void chkEndretDeltid_CheckedChanged(object sender, EventArgs e)
        {
            try
            {
                txtEndretProsent.Enabled = chkEndretDeltid.Checked;
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
        private void chkNyDeltid_CheckedChanged(object sender, EventArgs e)
        {
            try
            {
                txtNyProsent.Enabled = chkNyDeltid.Checked;
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }
        private void lagreAlt()
        {
            throw new NotImplementedException("Lagring er ikke implementert");
        }
    }
}
```

Student – entitetsklasse

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace StudentKursLibA
{
    public class Student
    {
        private int studnr = 0;
        private string studnavn = "";
        private List<Kurs> studkurs = new List<Kurs>();

        public Student(int studnr, string studnavn)
        {
            setStudnr(studnr);
            setStudnavn(studnavn);
        }
```

```
        public int getStudnr(){return studnr;}

        private void setStudnr(int studnr)
        {
            if (studnr <= 0) throw new Exception("Kan ikke sette studnr <= 0");
            this.studnr = studnr;
        }

        public string getStudnavn(){return studnavn;}

        internal void setStudnavn(string studnavn)
        {
            if (studnavn == null || studnavn == "")
                throw new Exception("Studentnavn kan ikke settes, det må ha verdi");
            this.studnavn = studnavn;
        }
        public override string ToString()
        {
            return "Heltidsstudent " + shortString() + " - " + strStudkurs();
        }
        protected string shortString() { return getStudnr().ToString() + " " +
getStudnavn(); }

        protected string strStudkurs()
        { //returnerer alle kursnumre som én streng
            string returstreng = "(" + studkurs.Count().ToString() + " kurs): ";
            foreach (Kurs k in studkurs)
            {
                returstreng += k.getKursID() + "  ";
            }
            return returstreng;
        }
       public override bool Equals(Object obj)
        {
            return obj is Student && this.studnr == ((Student)obj).getStudnr() ;
        }
        public override int GetHashCode()
        {
            return studnr;
        }
        public List<Kurs> getStudkurs() { return studkurs; }

        internal void addStudkurs(Kurs kurs)
        {
            if (kurs == null) throw new Exception("Kan ikke legge inn kurs som er null");
            if (studkurs.IndexOf(kurs) != -1) //finnes fra før
                throw new Exception("Kan ikke legge inn kurs - studenten er allerede
innmeldt i dette kurset");
            studkurs.Add(kurs);
        }
        internal void delStudkurs(Kurs kurs)
        {
            if (!studkurs.Remove(kurs))
                throw new Exception("Kan ikke slette kurs - studenten er ikke innmeldt i
dette kurset");
        }
        public Kurs findKurs(string kursID)
        {
            foreach (Kurs k in studkurs)
            {
                if (k.getKursID() == kursID) return k;
```

```
            }
            return null;
        }
        public int countStudKurs() { return studkurs.Count; }
    }
}
```

Deltidsstudent – Entitetsklasse

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace StudentKursLibA
{
    public class Deltidsstudent : Student
    {
        public float prosent = 0;

        public Deltidsstudent (int studnr, string studnavn, float prosent): base (studnr,
studnavn)
        {
            setProsent(prosent);
        }

        public float getProsent() { return prosent; }

        internal void setProsent(float prosent)
        {
            if (prosent <= 0 || prosent >= 100)
                throw new Exception("Kan ikke skape deltidsstudent - prosent må være >0 og
<100");
            this.prosent = prosent;
        }

        public override string ToString()
        {
            return "Deltidsstudent " + shortString() + " " + prosent.ToString () + "% - " +
strStudkurs ();
        }

        public override bool Equals(object obj)
        {
            return obj is Deltidsstudent && this.getStudnr() == ((Student)obj).getStudnr();
        }

        public override int GetHashCode()
        {
            return base.GetHashCode();
        }
    }
}
```

Kurs – Entitetsklasse

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace StudentKursLibA
{
    public class Kurs
```

```
    {
        private string kursID;
        private String kursnavn;
        private List<Student> kursdeltakere = new List<Student>();

        public Kurs(string kursID, string kursnavn)
        {
            setKursID(kursID);
            setKursnavn(kursnavn);
        }

        public string getKursID(){return kursID;}

        private void setKursID(string kursID)
        {
            if (kursID == null || kursID == "") throw new Exception("Kan ikke sette kursID -
må ha verdi");
            this.kursID = kursID;
        }

        public string getKursnavn() { return kursnavn; }

        internal void setKursnavn (string kursnavn){
            if (kursnavn == null || kursnavn == "") throw new Exception("Kan ikke sette
kursnavn - må ha verdi");
            this.kursnavn = kursnavn;
        }
        public override string ToString()
        {
            return "Kurs " + kursID + " " + kursnavn + strKursdeltakere();
        }
        private string strKursdeltakere()
        {
            string returstreng = " (" + kursdeltakere.Count().ToString() + " deltakere):";
            foreach (Student s in kursdeltakere)
            {
                returstreng += s.getStudnr().ToString() + " ";
            }
            return returstreng;
        }
        public override bool Equals(object obj)
        {
            return obj.GetType() == this.GetType() && this.kursID ==
((Kurs)obj).getKursID();
        }
        public override int GetHashCode()
        {
            return kursID.GetHashCode(); //bruker kursIDs hash som kursets hash
        }
        public List<Student> getKursdeltakere() { return kursdeltakere; }

        internal void addKursdeltakere(Student student)
        {
            if (student == null) throw new Exception("Kan ikke legge inn student som er
null");
            if (kursdeltakere.IndexOf(student) != -1) //finnes fra før
                throw new Exception("Kan ikke legge inn student - studenten er allerede
innmeldt i dette kurset");
            kursdeltakere.Add(student);
        }
        internal void delKursdeltakere(Student student)
        {
```

```
            if (!kursdeltakere.Remove(student)) //studenten finnes ikke
                throw new Exception("Kan ikke slette student - studenten er ikke innmeldt i 
dette kurset");
        }
        public Student findKursdeltakere(int studnr)
        {
            foreach (Student s in kursdeltakere)
                if (s.getStudnr() == studnr) return s;
            return null;
        }
        public int countKursdeltakere() { return kursdeltakere.Count; }
    }
}
```

StudentKursKontroll – Kontrollklasse

```
using System;
using System.Collections.Generic;

namespace StudentKursLibA
{
    public class Kontroll
    {
        //Denne klassen gjøres som en singleton med bare én instans, her kalt "kontroll"
        //Man får tilgang til den ved å skrive "Kontroll.kontroll"
        private List<Student> studenter = new List<Student>();
        private List<Kurs> kurser = new List<Kurs>();
        static public readonly Kontroll kontroll = new Kontroll();

        //Konstruktør
        private Kontroll() //private så ingen andre objekter kan bruke den
        {
        }

        //Metoder
        public List<Student> getStudenter()
        {
            return studenter;
        }

        public void addStudenter(Student student)
        {
            if (student == null) throw new Exception("Kan ikke legge til student uten 
verdi");
            if (findStudenter(student.getStudnr()) != null) throw new Exception("Kan ikke 
legge til studenten - finnes fra før");
            studenter.Add(student);
        }
        public void delStudenter(Student student)
        {
            if (student == null) throw new Exception("Kan ikke slette en student uten 
verdi");
            if (!studenter.Remove(student)) throw new Exception("Kan ikke slette studenten - 
finnes ikke");
            //Slett alle referanser til denne studenten i kurs
            foreach (Kurs k in student.getStudkurs())
                k.delKursdeltakere(student);
        }
        public Student findStudenter(int studnr)
        {
            foreach (Student s in studenter)
                if (s.getStudnr() == studnr) return s;
```

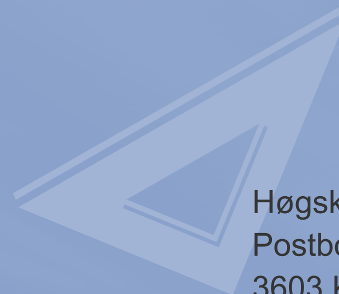
```
            return null;
        }
        public int countStudenter() { return studenter.Count; }

        public List<Kurs> getKurser() { return kurser; }

        public void addKurser(Kurs kurs)
        {
            if (kurs == null) throw new Exception("Kan ikke legge til kurs uten verdi");
            if (findKurser(kurs.getKursID()) != null) throw new Exception("Kan ikke legge
til kurset - det finnes fra før");
            kurser.Add(kurs);
        }
        public void delKurser(Kurs kurs)
        {
            if (kurs == null) throw new Exception("Kan ikke slette kurs uten verdi");
            if (!kurser.Remove(kurs)) throw new Exception("Kan ikke slette kurset - finnes
ikke");
            foreach (Student s in kurs.getKursdeltakere())
                s.delStudkurs(kurs);
        }
        public Kurs findKurser(string kursID)
        {
            if (kursID == null || kursID == "") throw new Exception("Kan ikke finne et kurs
uten kursID");
            foreach (Kurs k in kurser)
                if (k.getKursID() == kursID) return k; //funnet
            return null; //ikke funnet
        }
        public int countKurser() { return kurser.Count; }

        public void meldinn(int studnr, string kursID)
        {
            Student stud = findStudenter(studnr);
            Kurs kurs = findKurser(kursID);
            if (stud == null || kurs == null)
                throw new Exception("Finner ikke studenten eller kurset");
            stud.addStudkurs(kurs);
            kurs.addKursdeltakere(stud);
        }
        public void meldut(int studnr, string kursID)
        {
            Student stud = findStudenter(studnr);
            Kurs kurs = findKurser(kursID);
            if (stud == null || kurs == null)
                throw new Exception("Finner ikke studenten eller kurset");
            stud.delStudkurs(kurs);
            kurs.delKursdeltakere(stud);
        }
        public void endreStudent(int studnr, string studnavn,  bool deltid, float prosent =
0)
        {
            Student endretStudent = findStudenter(studnr);
            Student nyStudent;
            if (endretStudent == null) throw new Exception("Finner ikke studenten");
            //For å få enklere kode, erstattes forrige student med en ny student uansett
            //selvom det gir tyngre eksekvering, særlig i større systemer
            if (deltid)
                nyStudent = new Deltidsstudent(studnr,studnavn,prosent);
            else
                nyStudent = new Student(studnr,studnavn);
            foreach (Kurs k in endretStudent.getStudkurs()) //få med alle koblinger
```

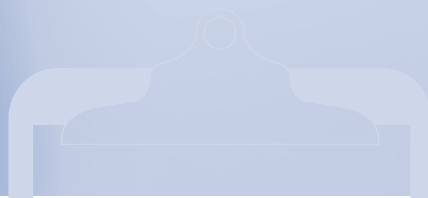
```
                nyStudent.addStudkurs(k);
            studenter.Remove(endretStudent);
            studenter.Add(nyStudent);
          }
        public void endreKurs(string kursID, string kursNavn)
        {
            Kurs endretKurs = findKurser(kursID);
            if (endretKurs==null) throw new Exception("Finner ikke kurset");
            endretKurs.setKursnavn(kursNavn);
        }
        #region Demo A - Ingen persistens
        public void lesData(string connString)
        {
            throw new NotImplementedException("Ikke implementert lesing av objekter");
        }
        public void lagreData(string connString)
        {
            throw new NotImplementedException("Ikke implemtert lagring av data");
        }
        #endregion
    }//Kontroll
}
```



Høgskolen i Buskerud
Postboks 235
3603 Kongsberg
Telefon: 32 86 95 00

www.hibu.no

ISSN 0807-4488 (trykt)
ISSN 1893-2312 (online)



HØGSKOLEN
i Buskerud