UNIVERSITETET I OSLO

Objektdatabaser

ODMGs objektmodell

og

ODL (Object Definition Language)

Databaser vs objektorientering

- Det er en inherent (iboende) konflikt mellom objektorientering og objektdatabasesystemer
- I objektorientering er innkapsling et sentralt begrep: Den eneste mekanismen som finnes for å la et objekt få vite noe om hvordan et objekt fra en annen klasse fungerer, er arv
- Objektdatabaser er databaser, og de skal ha et begrepsmessig skjema
- Dermed trenger objektdatabaser noe i tillegg til arv
- Det viktigste er en mekanisme som erstatter relasjons-databasenes fremmednøkler, men en slik mekanisme passer ikke inn i «ren» objektorientering

OMG og ODMG

- Derfor har verden to industri- og universitetsoppnevnte standardiseringskomiteer for objektorientering
- OMG (Object Management Group) forvalter en de facto standard for hva objektorientering er
- OMG har klassisk objektorientering som sitt mantra, og de har ikke villet å gå på akkord for å tilfredstille databaseindustrien
- ODMG (Object Data Management Group) forvalter en standard for objektdatabaser som tilfredstiller databaseindustriens krav, men som ellers ligger så nær opp til OMG-standarden som mulig

Viktige OMG-standarder

- **CORBA** (Common Object Request Broker Architecture)
  - Arkitektur for mellomvare mellom (objektorienterte) klienter og (objektorienterte) tjenere
  - Omfatter språket IDL (Interface Definition Language) som brukes til å spesifisere objektklasser
- **UML** (Unified Modelling Language)
  - Er på godt og vondt i ferd med å bli et enerådende sett av modelleringsspråk for objektorienterte systemer
  - Er på rask fremmarsj både i industri og undervisning
- Hjemmesiden for OMG er http://www.omg.org/

ODMG standarden – I

- ODMG (Object Data Management Group) forvalter en standard for objektdatabaser bestående av
  - En objektmodell som ligger så nær opp til OMG-standarden som mulig
  - ODL (Object Definition Language) som er et språk for å definere objektklasser (OMGs IDL er inneholdt i ODL)
  - OIF (Object Interchange Format) som er et format for å skrive og lese objekter til og fra fil
  - OQL (Object Query Language) som er et spørrespråk hvis syntaks med vilje er gjort mest mulig lik SQL
  - Språkbindinger for Smalltalk, C++ og Java (Simula er for gammel og C# for ung for at bindinger for disse språkene er med i standarden)

ODMG standarden – II

- Merk at standarden ikke har noe datamanipuleringsspråk
  - all oppdatering skal foregå via et vertsspråk
- Hovedhensikten med utvekslingsformatet OIF er å kunne utveksle objekter mellom ulike OODBMSer
- Gjeldende ODMG standard er ODMG 3.0 (2000)
- Der er ingen av språkbindingene fullstendig definert, så vi venter på en ny versjon av standarden
- Hjemmesiden for ODMG er http://www.odmg.org/

Elementer i ODMGs begrepsverden

- Objektidentitet og objektidentifikator (OID)
- Objekter og verdier (verdier er ikke objekter)
- Ekstensjon (Extent) – instansene i en klasse
- Komplekse objekter og typekonstruktører
- Operatorer
- Programmeringsspråkkompabilitet (sømløs overgang mellom database og applikasjons-programmer)
- Innkapsling (skjulte data)
- Type- eller klassehierarkier med arv og polymorfisme

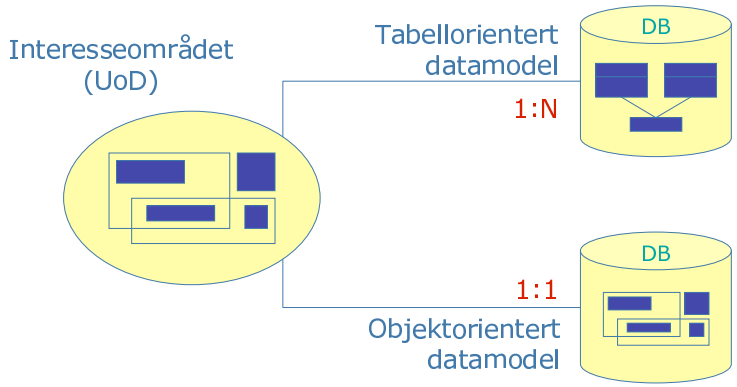
Det objektorienterte paradigme Klassifikasjon og taksonomi

- Paradigme: OO er laget for å kunne modellere et interesseområde (ofte kalt miniverden eller UoD) som en samling av kommuniserende og samarbeidende entiteter (enheter) som vi kaller ***objekter***
- Klassifikasjon: Objektene grupperes i klasser som er en
  - felles beskrivelse (mal) for alle objekter som tilhører samme klasse, kalt klassens ***intensjon*** (intent)
  - samling av objekter med felles egenskaper, kalt klassens ***ekstensjon*** (extent)
- Taksonomi: Klassene ordnes i super- og subklasser Dette gir opphav til arv av egenskaper og polymorfisme

Abstraksjon og autonomi

- Objekt: <verdi, {operatorer}>
  - Objektene er distinkte og universelt identifiserbare
  - Operatorene er realisert (implementert) som metoder
- Verdi: Datastruktur
  - Verdier kan være forskjellig fra andre verdier, men ikke distinkte og universelt identifiserbare
- Innkapsling:
  Et objekt kan inneholde og skjule intern informasjon (Forutsetter at objektene oppfører seg «pent» og ikke «snoker i andres interne data»)
- Kontrakt:
  Et krav om at objekter oppfører seg (kommuniserer og samarbeider) i henhold til avtalte regler

Målet for OO-datamodellering: 1:1-modell av interesseområdet



OO-modellering og objektdatabaser

- Sammenlignet med tradisjonelle datamodeller skal en OO-modell gi
  - Høy modularitet gjennom meningsfulle abstraksjoner
  - Bedre kontroll med kompleksitet
  - Klar avgrensning mellom klassen som abstrakt datatype (ADT) og dens implementasjon
- OO-modellering skal fremme evolusjonær systemdesign med inkrementell programmering og gjenbruk
- En objektdatabase (ODB) er en persistent samling av objekter
- Et ODB-objekt har formatet <OID, verdi, {operasjoner}>
- Et OODBMS er et DBMS som understøtter ODBer

ODB-eksempel

- Eksempelklasse:

```
class  Konto  {
       integer          ktonr;
       real             saldo;
       REF Kunde        eier;
}
```

- Eksempelverdier:
  - V_1 = tuppel (ktonr: 65536, saldo = 34567.50, eier = $\hat{}K_1$)
  - V_2 = tuppel (ktonr: 17289, saldo = 24506.52, eier = $\hat{}K_2$)
  - V_3 = tuppel (ktonr: 87654, saldo = 21451.07, eier = $\hat{}K_1$)
- Eksempelobjekter:
  - O_1 = <•, V_1, •>
  - O_2 = <•, V_2, •>
  - O_3 = <•, V_3, •>

Krav til OODBMSer i "The OODB Manifesto" (1990)

Må ha

- OID (objektidentifikator)
- Komplekse og sammensatte objekter
- Brukerdefinerte typer
- Beregningskomplett språk
- Innkapsling
- Arv med type/klasse-hierarkier
- Polymorfisme: overlasting, redefinisjoner, sen binding

... Alle er ortogonale egenskaper

Bør ha

- Objektversjonering
- Støtte for distribusjon
- Nye transaksjonsmekanismer
- Støtte for regelbaserte systemer (aktive og deduktive Dber)

... og mye mer

OID – I

- Objekter eksisterer uavhengig av sine (nåværende) verdier
  - Uansett hvordan verdiene i et objekt endres, er objektet det samme
  - Objekter identifiseres entydig av sin OID
  - Du får aldri feil objekt hvis objektet refereres via sin OID
- Begrepene ***identitet*** og ***likhet*** eksisterer begge, og de betyr ikke det samme
  - At to objekter er identiske, betyr at de er samme objekt, dvs. at de har samme OID
  - At to objekter er like, betyr at de tilhører samme klasse og har de samme verdiene
- En OID baseres aldri på foranderlige (mutable) data

OID – II

- En OID er (i praksis) alltid systemgenerert og -håndtert
- OIDer er unike (systemvidt, globalt og universalt)
  - GUID: Globally Unique ID (egenskap i MS-verden)
  - UUID: Universally Unique ID (egenskap i Unix-verden)
- En OID er uforanderlig (immutable) gjennom hele objektets liv, og den gjenbrukes ikke etter objektets død

- Noen eksempler på objektoperasjoner basert på OIDer:
  - Å sammenligne objekter for identitet, likhet o.l.
  - Å referere objekter
  - Å finne og hente objekter

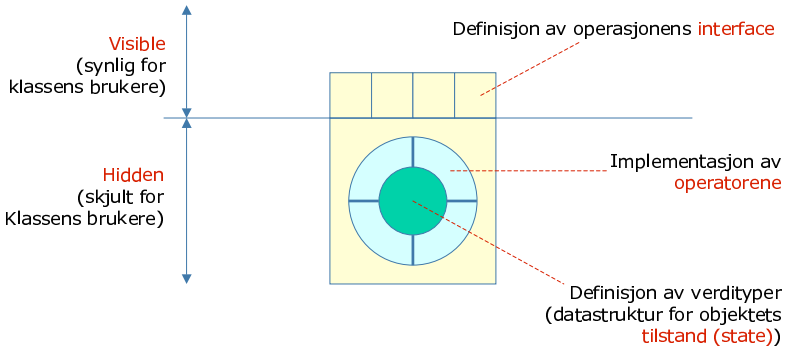
Komplekse og sammensatte objekter

- OO-paradigmet krever støtte for komplekse objekter
- Det er to typer kompleksitet som støttes
  - Ustrukturerte komplekse objekter
    Eksempler er tidsserieobjekter, fritekstobjekter og medieobjekter (lyd, bilde, video)
    Vi deler dem inn i to grupper:
    - BLOBs (Binary Large Objects)
    - CLOBs (Character Large Objects)
  - Strukturerte komplekse objekter
    Det samme som sammensatte objekter, objekter som inneholder andre objekter eller deler av andre objekter

Klasser og typer

- Det er ikke full internasjonal enighet om terminologien
- I Skandinavia (og i det meste av verden) bruker vi ordene slik:
  - En type, eller mer presist, en abstrakt datatype (ADT) er intensjonen til en klasse
  - En klasse er en implementasjon av en type (ADT)
  - Et objekt er en instansiering av en klasse (og ikke av en ADT)
- Brukerne kan lage sine egne klasser og typer
- Oppsummering:
  - Et objekt har en type og er instans av en klasse

Innkapsling



Arv og type- og klassehierarkier

- Vi har to typer arv vi må skille mellom:
  - Typearv hvor en ADT (subtypen) arver egenskaper fra en annen ADT (supertypen)
  - Klassearv hvor en klasse (subklassen) arver implementasjon og objekter fra en annen klasse (superklassen)
- Vi skiller mellom
  - Enkel arv som leder til et type- eller klassehierarki
  - Multippel arv som leder til en typelattice (multippel arv for klasser fører til store problemer og er vanligvis ikke tillatt)

Polymorfisme

Polymorfisme har tre fasetter:

- Overlasting
  Bruk av samme navn på forskjellige operatorer (i forskjellige ADTer)
- Redefinisjon
  Reimplementasjon av operatorer lengre ned i klassehierarkiet
- Sen binding
  Binde et operatornavn til en spesifikk implementasjon dynamisk under “run-time”
  (gjøres individuelt for hvert objekt)

Oversikt over begrepsapparatet i ODL

I Object Definition Language (ODL) defineres:

- Klasser (**`class`** og **`interface`**)
- Attributter (**`attribute`**)
- Assosiasjoner (**`relationship`**)
- Metoder
- Typesystemet
- Ekstensjoner (**`extent`**)
- Nøkler (**`key`** eller **`keys`**)
- Arv

Klassedeklarasjoner

- En ODL-klassedefinisjon består av et klassehode fulgt av en kropp omsluttet av krøllparenteser
- Klassehodet består av klassens navn og en parentes med navn på ekstensjonen og eventuelle nøkler
- Eksempel:

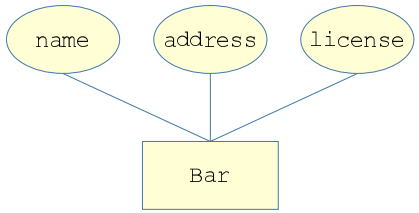
```
class Student (extent studenter keys (fdato, personnr), brukernavn)
{ <klassekroppen>
}
```

Man ***må*** ikke navngi ekstensjonen, men det er nødvendig for å kunne bruke den i spørsmål
Man kan angi null eller flere nøkler (to i eksemplet)

Attributter – I

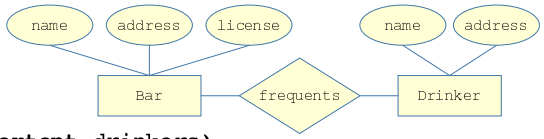
- Syntaks:
  `attribute <attribute-type> <attribute-elements>`
- Exempel:

```
class Bar (extent bars)
{  attribute string name;
   attribute Struct addr
        {       string street,
                string city
        }       address;
   attribute Enum license
        {       full,
                beer,
                none
        }       licenseType;
}
```



Attributter – II

- Vi gir **`Struct`**-er and **`Enum`**-er navn fordi vi trenger å referere til dem
- Eksempel:




```
class Drinker (extent drinkers)
{   attribute string name;
    attribute Struct Bar::addr address
}
```

- MERK: Vi gjenbruker Struct-en addr i Bar som type for address-attributtet i Drinker
- Elementer i en annen klasse refereres ved klassenavnet og et dobbelt kolon: <class-name>::<element-name>

Assosiasjoner – I

- Assosiasjoner (relationships) forbinder objekter med hverandre
- Assosiasjoner opptrer alltid i par:
  Hvis A er assosiert med B, må B være assosiert med A

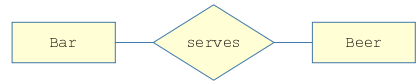
- Syntaks:
  ```
  relationship <relationship-type> <relationship-name>
  inverse <class-name>::<inverse-relationship-name>
  ```

- Eksempler:
  ```
  relationship Set<Person> hasKids
  inverse Person::hasParents;
  relationship Person hasSpouse inverse Person::hasSpouse;
  relationship Set<Car> hasCars inverse Car::hasOwner;
  ```


Assosiasjoner – II

Eksempel:

Sammenhengen `serves` mellom `Bar` og `Beer` er representert med en assosiasjon i `Bar` som gir deg de ølmerker som selges der, og en invers assosiasjon `servedAt` i `Beer` som angir de barene som selger dette ølet

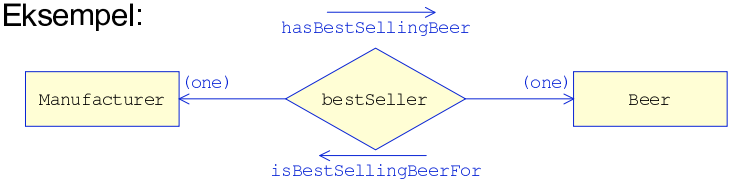



```
class Bar
{ relationship Set<Beer> serves inverse Beer::servedAt;
}

class Beer
{ relationship Set<Bar> servedAt inverse Bar::serves;
}
```


Assosiasjoner – III

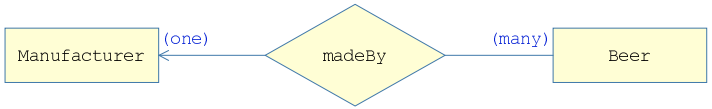
- 1:1 assosiasjoner realiseres med to enkle referanser (pekere)
- Eksempel:




```
class Manufacturer
{   relationship Beer hasBestSellingBeer
    inverse Beer::isBestSellingBeerFor;
}
class Beer (extent beers)
{   relationship Manufacturer isBestSellingBeerFor
    inverse Manufacturer::hasBestSellingBeer;
}
```


Assosiasjoner – IV

- 1:n assosiasjoner realiseres med en enkel referanse på mange-siden og en mengde referanser på én-siden
- Eksempel:



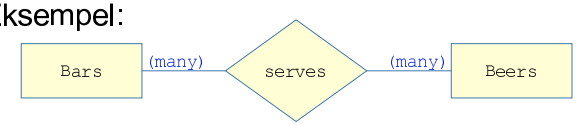

```
class Manufacturer
{   relationship Set<Beer> makes inverse Beer::madeBy;
}

class Beer
{   relationship Manufacturer madeBy
    inverse Manufacturer::makes;
}
```

MERK:
Assosiasjonsnavnene følger leseretningen:
"`Manufacturer` **`makes`** `Beer`" fra `Manufacturer` til `Beer`, og
"`Beer` **`madeBy`** `Manufacturer`" fra `Beer` til `Manufacturer`

Assosiasjoner – V

- m:n assosiasjoner realiseres med to mengder referanser, en på hver side
- Eksempel:

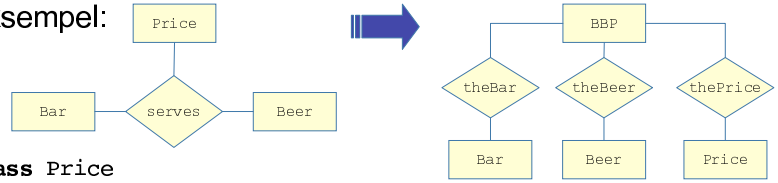



```
class Bar
{   relationship Set<Beer> serves inverse Beer::servedAt;
}

class Beer
{   relationship Set<Bar> servedAt inverse Bar::serves;
}
```


Assosiasjoner – VI

- ODL tillater bare binære assosiasjoner
- Ternære (3-veis) og høyere ordens assosiasjoner må implementeres ved hjelp av en «kryssreferanseklasse»
- Eksempel:




```
class Price
{   attribute real price;
    relationship Set<BBP> toBBP inverse BBP::thePrice
}
class BBP
{   relationship Bar theBar inverse …;
    relationship Beer theBeer inverse …;
    relationship Price thePrice inverse Price::toBBP;
}
```


Metoder – I

- En metode er et navngitt og parametrisert stykke eksekverbar kode (prosedyre, funksjon) som fungerer som en operator i klassens objekter
- Bare metodesignaturen (dvs. metodenavnet og mengden av parametere med tilhørende typer) er en del av ODL
- Selve koden (realiseringen) skrives i vertsspråket (Tillatte vertsspråk er Java, C++ og Smalltalk)
- En metode kan returnere en verdi, og den kan kaste unntak (raise exceptions)
- Alle metodeparametre og en eventuell returverdi må være typet

Metoder – II

- I tillegg til at parametrene skal være typet, skal de merkes som `in`, `out` eller `inout`:
  - `in` – gir en kopi av parameterverdien til metoden (metoden kan ikke forandre den aktuelle parameteren)
  - `out` – for å gi verdier ut fra metoden
  - `inout` – begge de ovenstående, men gir en referanse (og ikke verdien) til den aktuelle parameteren
- To likeverdige eksempler:

```
class Bar
{   . . .
    void availableBeers(out Set<Beer>);
    . . .
}
```

```
class Bar
{   . . .
    Set<Beer> availableBeers();
    . . .
}
```

Typesystemet – I

- Basistyper:
  - **integer, real, float, boolean, character**
  - **string**
  - **time, date, interval**
  - **enumerated types**
  - … og fler
- Typekonstruktør:
  - **Struct** <navn> {<type><navn>, <type><navn>, … } definerer en type som er en struktur satt sammen av type-og-navn-par, lik en Cobol- eller Pascal-record

Typesystemet – II

- Samlinger (Collection types):
  - **Set<T>** uordnet mengde av (distinkte) objekter av type T
  - **Bag<T>** uordnet samling av objekter av type T hvor duplikater er tillatt
  - **List<T>** ordnet samling av objekter av type T hvor duplikater er tillatt
  - **Array<T>** ordnet og indeksert samling av objekter av type T hvor duplikater er tillatt
  - **Dictionary<S,T>** mengde av objektpar av type S og T
- Merk: Typen til en assosiasjon (relationship) må være en klasse eller en samling av klasser

Klasser og ekstensjoner

- Når man har definert en klasse med nøkkelordet **`class`**, kan man generere objekter av denne klassen ved å bruke nøkkelordet **`new`**
- Hvert objekt tildeles sin unike OID, og mengden av genererte objekter utgjør klassens ekstensjon
- I ODL kan man navngi ekstensjonen med nøkkelordet **`extent`**
- Eksempel:

```
class Student (extent studenter key fnr)
{ ...
}
```

- Det er nødvendig å navngi ekstensjonen for å kunne bruke OQL direkte mot den
- Eksempel:

```
select ... from studenter where ...
```


Interfacer – I

- En interface er en klasse det ikke kan lages objekter av
- Interfacer defineres ved å bruke nøkkelordet **`interface`** i stedet for **`class`**
- Siden de ikke kan lages instanser av, er det meningsløst (og ulovlig) å bruke nøkkelordene **`extent`** og **`key`** (eller **`keys`**) i interfacer
- Ellers er det ingen forskjell mellom definisjonen av en klasse og en interface
- Interfacer er nyttige når vi har flere ekstensjoner med (noen) felles egenskaper
- Vi lar da flere klasser arve samme interface
  Disse klassene arver da de attributtene, assosiasjonene og metodene som er definert i interfacen
- Her følger et eksempel som viser at både studenter og lærere er personer:

Interfacer – II

- Eksempel

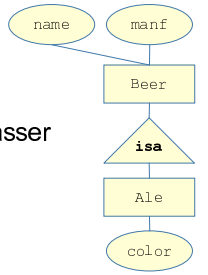
```
interface Person
{  attribute integer ssn;
   attribute string name;
   attribute date born;
   integer age(){... };
   ...
}

class Student : Person (extent students key ssn)
{  ...
}

class Teacher : Person (extent teachers key ssn)
{  ...
}
```


Arv – sub- og superklasser

- En subklasse arver alle egenskapene til sin superklasse
- Eksempel:
  `Ale` får alle attributter, assosiasjoner og metoder til klassen `Beer`



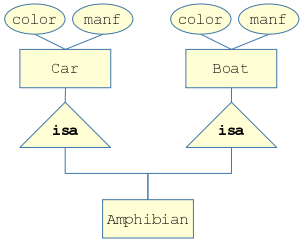
- Superklasser angies ved å prefikse dem med:
  - Kolon (:) for interfacer
  - Nøkkelordet **`extends`** for instansierbare klasser
- Eksempel: Ale er øl med farge:

```
class Ale extends Beer
{  attribute string color;
}
```

- Interfacer kan bare arve fra andre interfacer, mens klasser kan arve fra både klasser og interfacer

Multippel arv

- Multippel arv angis med en kolon-separert liste av de interfacene man arver fra
- Det første leddet i listen kan være en instansierbar klasse, og den må da prefikses med nøkkelordet **`extends`**
- Navnekonflikter er ikke tillatt, og det er programmererens ansvar å unngå dem



- Eksempel:
  `class` `Amphibian` **`extends`** `Car:Boat` **`{ ... }`**
  (Her er altså `Car` en klasse og `Boat` en interface)

ODL som relasjonsspesifikasjonsspråk

Utgangspunktet er at hver klasse blir til en relasjon.

Dette byr på mange problemer. Her er noen:

- I klasser uten nøkkel må man legge inn et fiktivt nøkkelattributt (en erstatning for OID)
- 1:n assosiasjoner kan oversettes til fremmednøkler, men m:n assosiasjoner må reliseres som egne relasjoner
- Rett frem oversettelse av struct-er og samlinger (som Set og Bag) gir lett opphav til dårlig normaliserte tabeller (Læreboken har eksempler på oppdateringsanomalier som kan oppstå på grunn av dette)

Konklusjon:
Unngå bruk av ODL hvis hensikten er å designe en relasjonsdatabase!