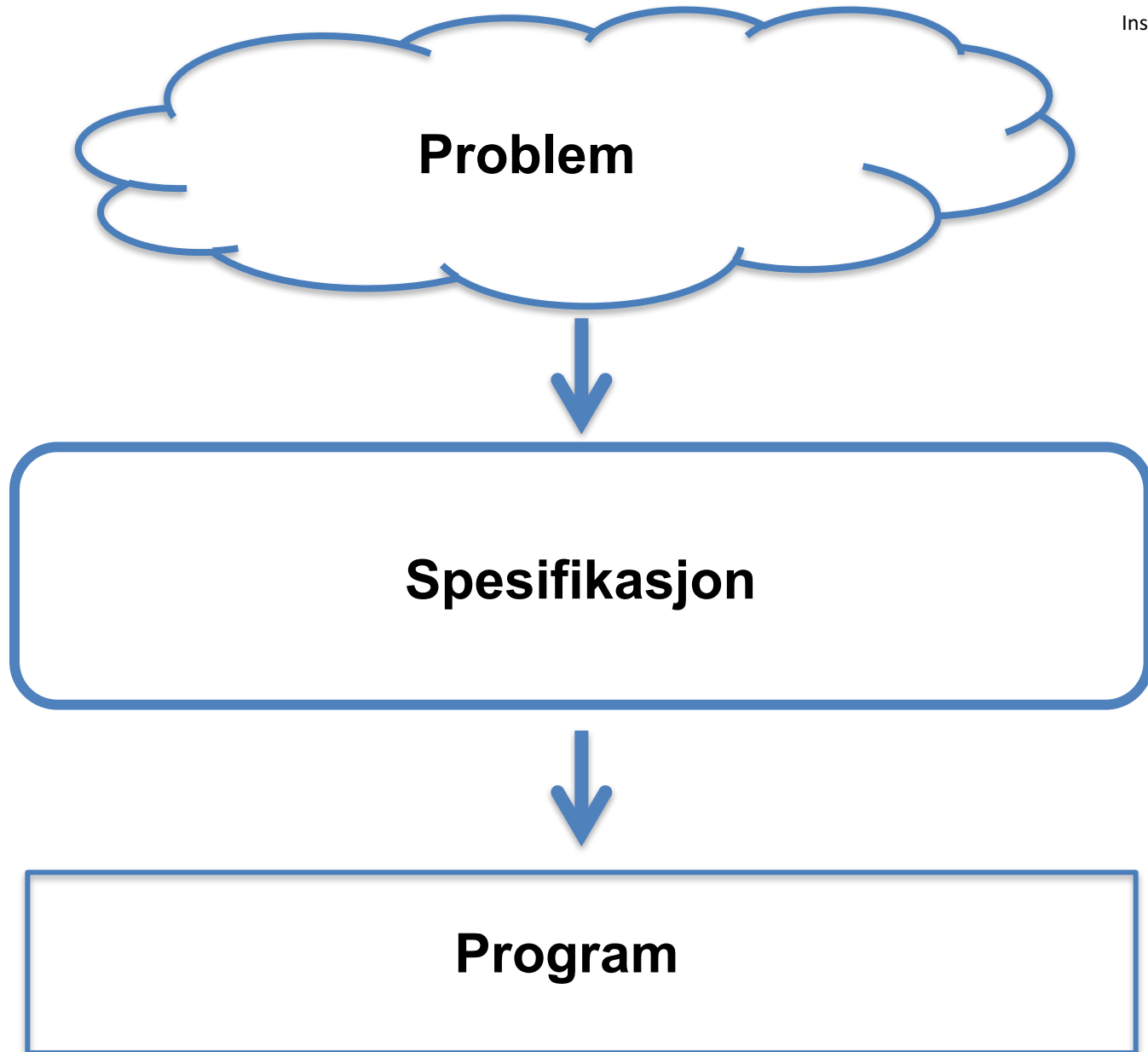




INF1010 - 20. april 2017

Programmeringsmønstre – Patterns Eksempel på bruk av interface

Stein Gjessing
Institutt for informatikk
Universitetet i Oslo



Programvare-arkitektur

- Hvordan programmet er bygget opp
 - Klasser
 - Objekter
 - Hvordan disse klassene og objektene er koblet sammen
 - Løs kobling
 - Tett kobling
 - Komponenter
 - Mapper, “packages”
 - . . .



Kjente mønstre for programoppdeling

- Det er mange kjente og velprøvde måter å dele opp programkode. Slike kjente måter kalles ”**design patterns**”.
 - Tatt fra “The Timeless Way of Building”, Christoffer Alexander, Oxford University Press, 1979, ISBN 0195024028
 - “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such away that you can use this solution a million times over, without doing it the same way twice”

Et slikt **programmeringsmønster** har mange lært i INF1000 med kommandoløkka som skriver ut en meny og tar kommando fra brukeren.

Arkitekturmønstre



Snøhettas forslag til
regjeringskvartal



Typisk hus i by i USA

Et eksempel på et kjent
programmeringsmønster:

Modell - Utsyn - Kontroll
(MVC - Model, View, Control)
(funnet på av Trygve Reenskaug)



Et bankprogram

- Vi skal lage et program som håndterer kontoene i en bank. En konto eies av en kunde, og har en saldo.
- Programmet skal kommunisere med en bruker i Unix/DOS-vinduet
- Programmet skal kunne administrere kontoene i banken, og i første omgang
 - 1) Opprette en ny konto
 - 2) Fjerne en konto
 - 3) Sette inn penger på en konto
 - 4) Skrive ut bankens forvaltningskapital

Et bankprogram

- Vi skal lage et program som håndterer kontoene i en bank. En konto eies av en kunde, og har en saldo.

Vi bruker substantivmetoden til å finne forslag til klasser (som vi kan lage objekter fra)

```
class Bank  
class Konto  
class Kunde  
class Saldo
```

For kunde og saldo satser vi i denne enkle versjonen av programmet på å bare bruke h.h.v. en “String” og en “double”

class Bank

- I en bank skal vi kunne legge inn en ny kunde, fjerne en kunde, sette inn penger på en kundes konto og finne forvaltningskapitalen til banken (summen av alle beløpene på alle kontoene)

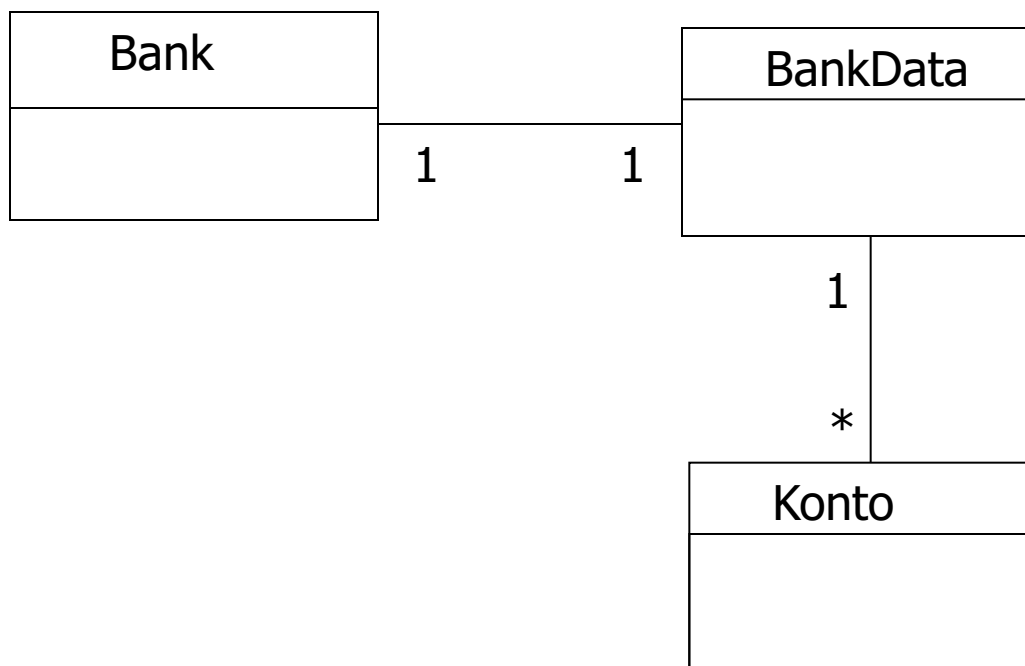
class BankData

- Alle kontoene blir adminstrert av klassen BankData

class Konto

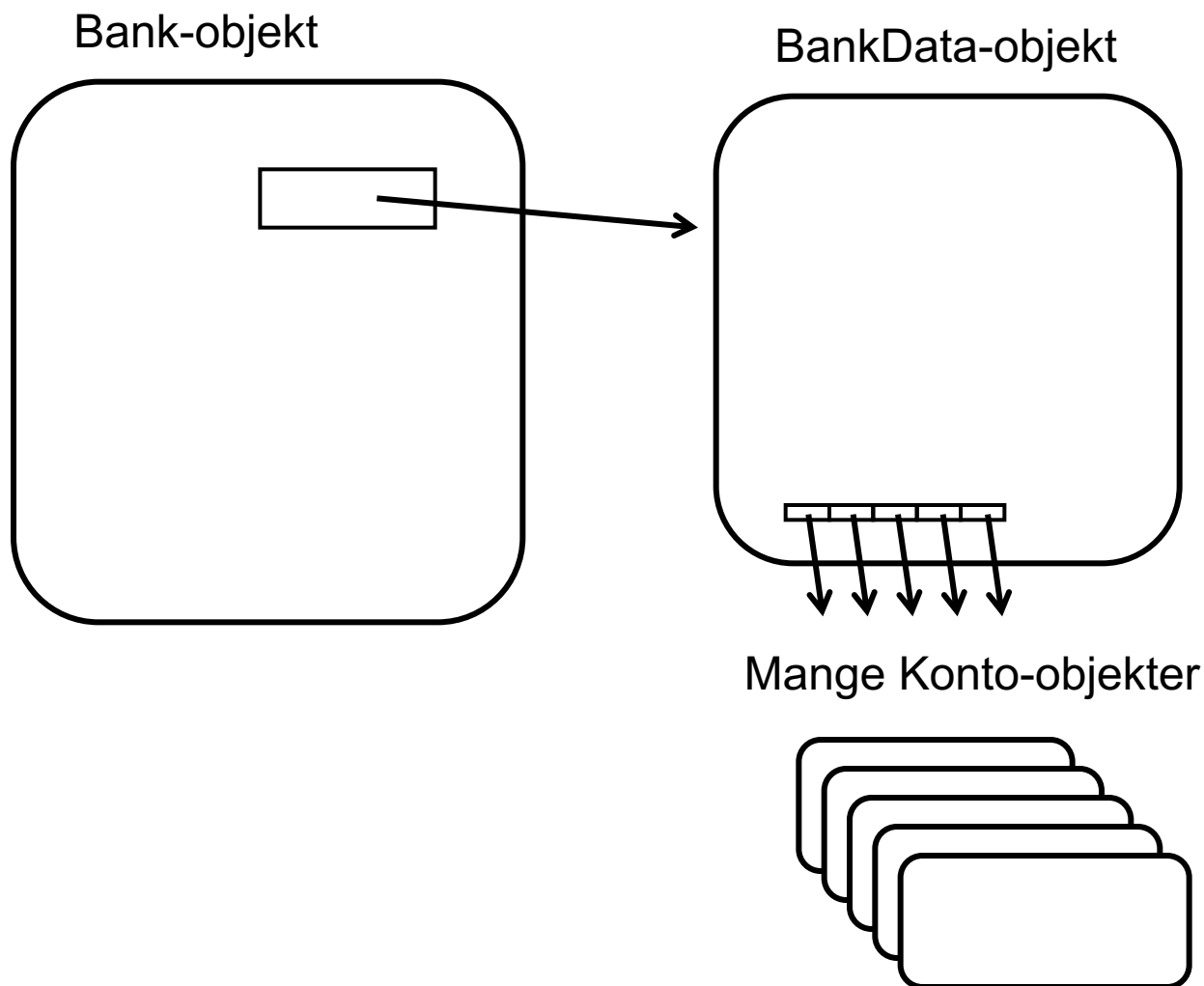
- Data og metoder for en konto

UML – klassediagram:

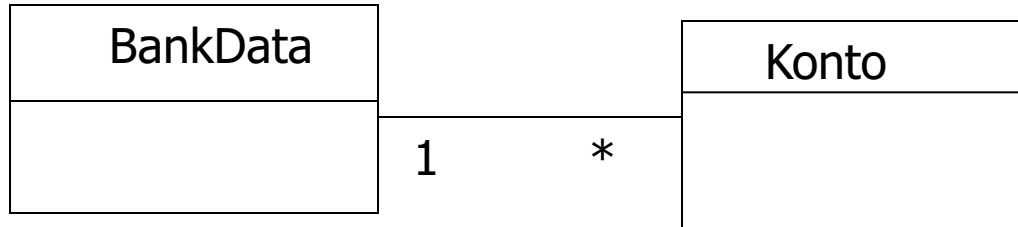


Sammenlign med Java datastrukturen på neste side
(UML er ikke viktig i INF1010)

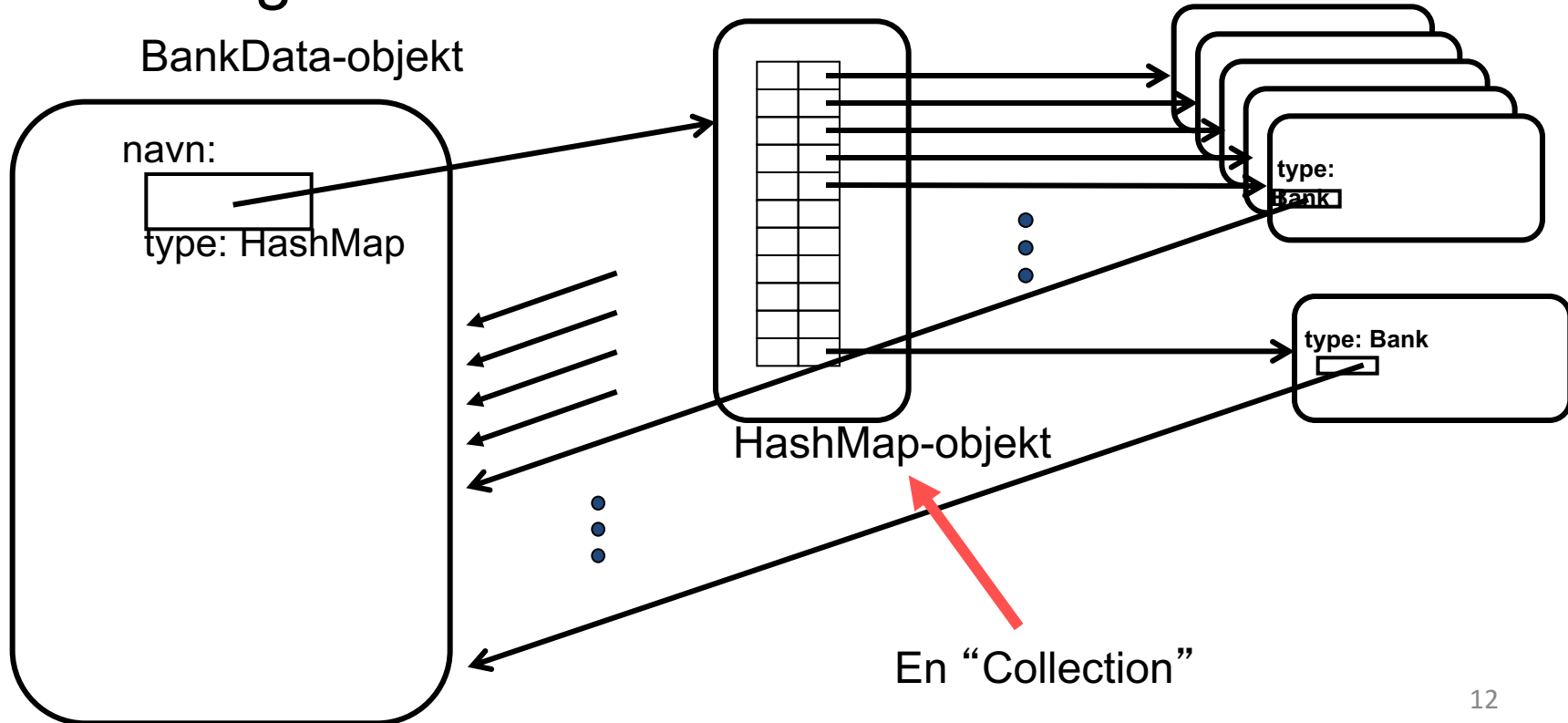
Java datastruktur for banksystemet



UML – klassediagram for BankData

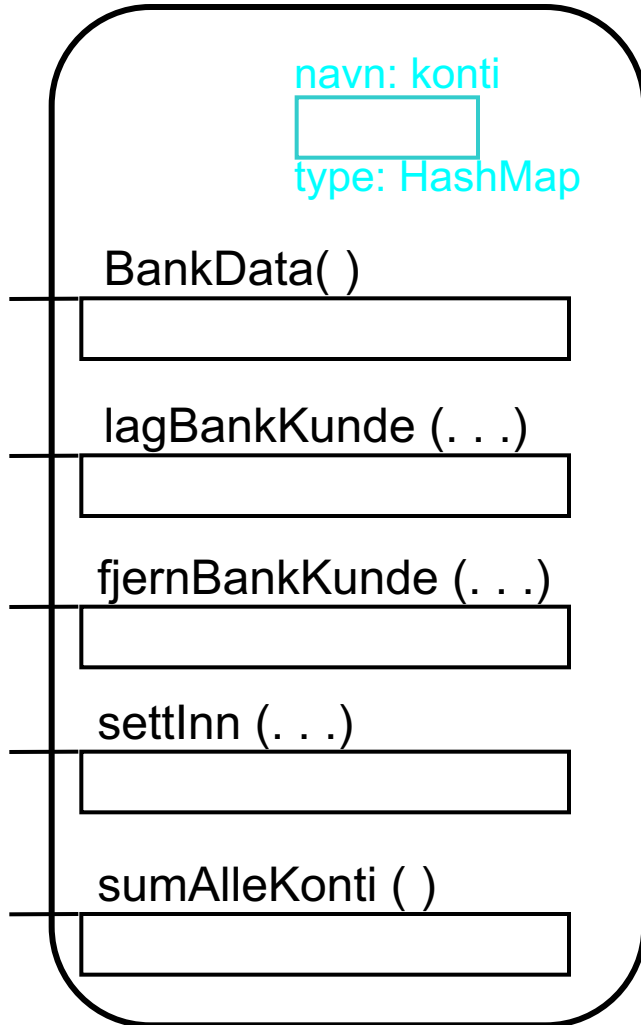


Kan f.eks. gi denne Java datastrukturen: Konto-objekter



Skisse av class BankData

BankData-objekt



```
class BankData {
```

```
    BankData( ) { . . . }
```

```
    private HashMap <String,Konto> konti =  
        new HashMap<String,Konto>( );
```

```
    public void lagBankKunde(String navn) {  
        . . . }
```

```
    public void fjernBankKunde (String n) {  
        . . . }
```

```
    public void settInn(String n, double b) {  
        . . . }
```

```
    double sumAlleKonti( ) {  
        . . . }
```

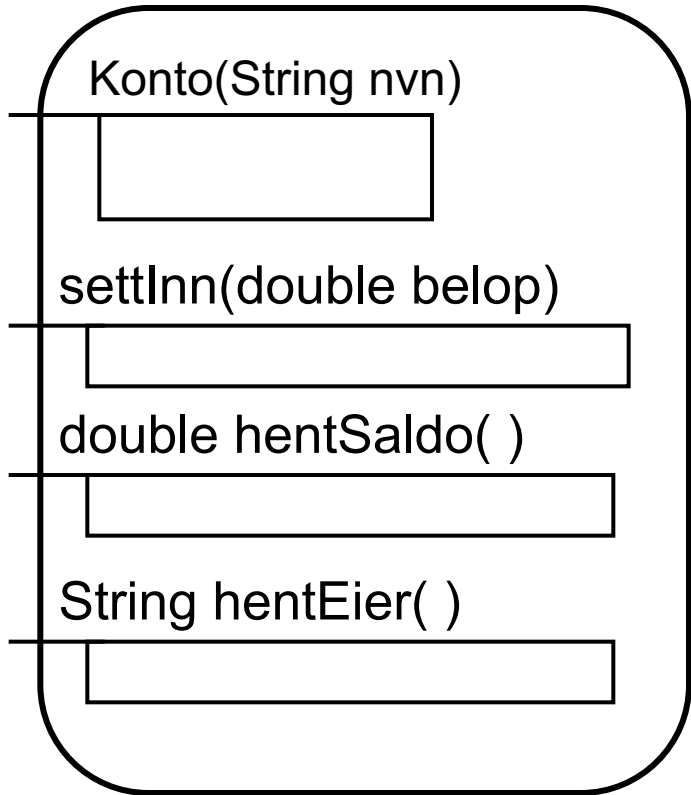
```
}
```

class Konto

- Når en konto opprettes oppgir vi dens eier, og saldoen settes til null.
- På en konto skal vi kunne sette inn et beløp, finne navnet på eieren og finne innestående beløp

Skisse av class Konto

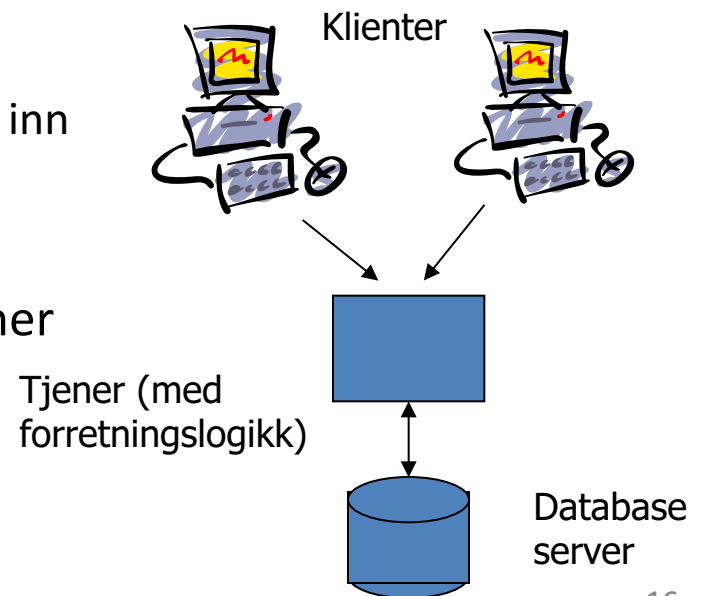
Konto-objekt



```
class Konto {  
  
    Konto (String nvn) { . . . }  
  
    public void setlInn(double belop) { . . . }  
  
    public double hentSaldo ( ) { . . . }  
  
    public String hentEier( ) { . . . }  
  
} // slutt klassen Konto
```

Modell – Utsyn – Kontroll = MVC: Model – View – Control

- Modell
 - En ren datastruktur som holder alle data om problemet.
- Utsyn
 - Egen modul for å presentere modellen for brukeren på ulike måter valgt av brukeren (via kontrollen)
- Kontroll
 - Egen modul/klasse for å motta ordre fra brukeren og enten endre modellen (legge inn nye data, endre, fjerne data) eller kalle en funksjon i utsynet for å gi et nytt bilde av modellen.
- Omlag samme oppdeling som i klient/tjener baserte distribuerte systemer
 - Modell – database (server)
 - Utsyn – klienten på distribuerte PCer
 - Kontroll – tjener/server med forretningslogikken





Modell - Utsyn - Kontroll (MVC)

- Her: Et svært enkelt system for å vise mekanismen. MVC er fortrukket ved noen hundre linjer kode og oppover.
- Grunnen til å skille disse funksjonene fra hverandre, er at det gjør det lettere :
 - Å utvikle - Konsentrere seg om en ting om gangen
 - Å endre - Kode om en ting er samlet ett sted
F.eks.: Linjebasert vs. Vindusbasert
utskrift

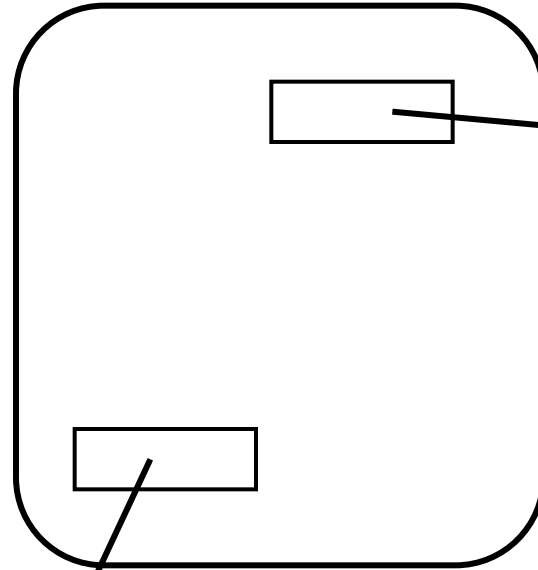


MVC – løsning for banken vår

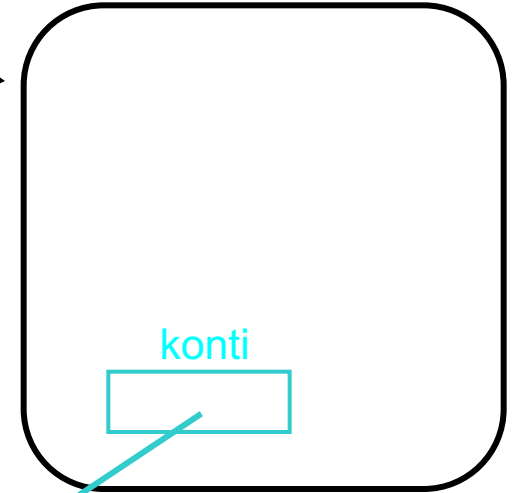
- Det skal da være klasser som håndterer :
 - Klassen **BankData**
 - med hjelpeklassen **Konto**
 - Utsynet for hele systemet (**BankUtsyn**)
 - Kontrollen for hele systemet (**Bank**)

Java datastruktur – oversikt

Bank-objekt



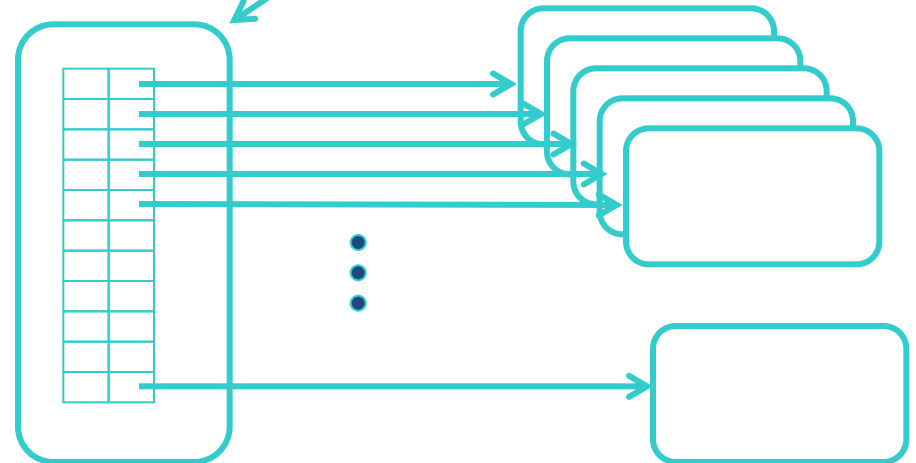
BankData-objekt



BankUtsyn-objekt



Konto-objekter



Program- merings- mønsteret

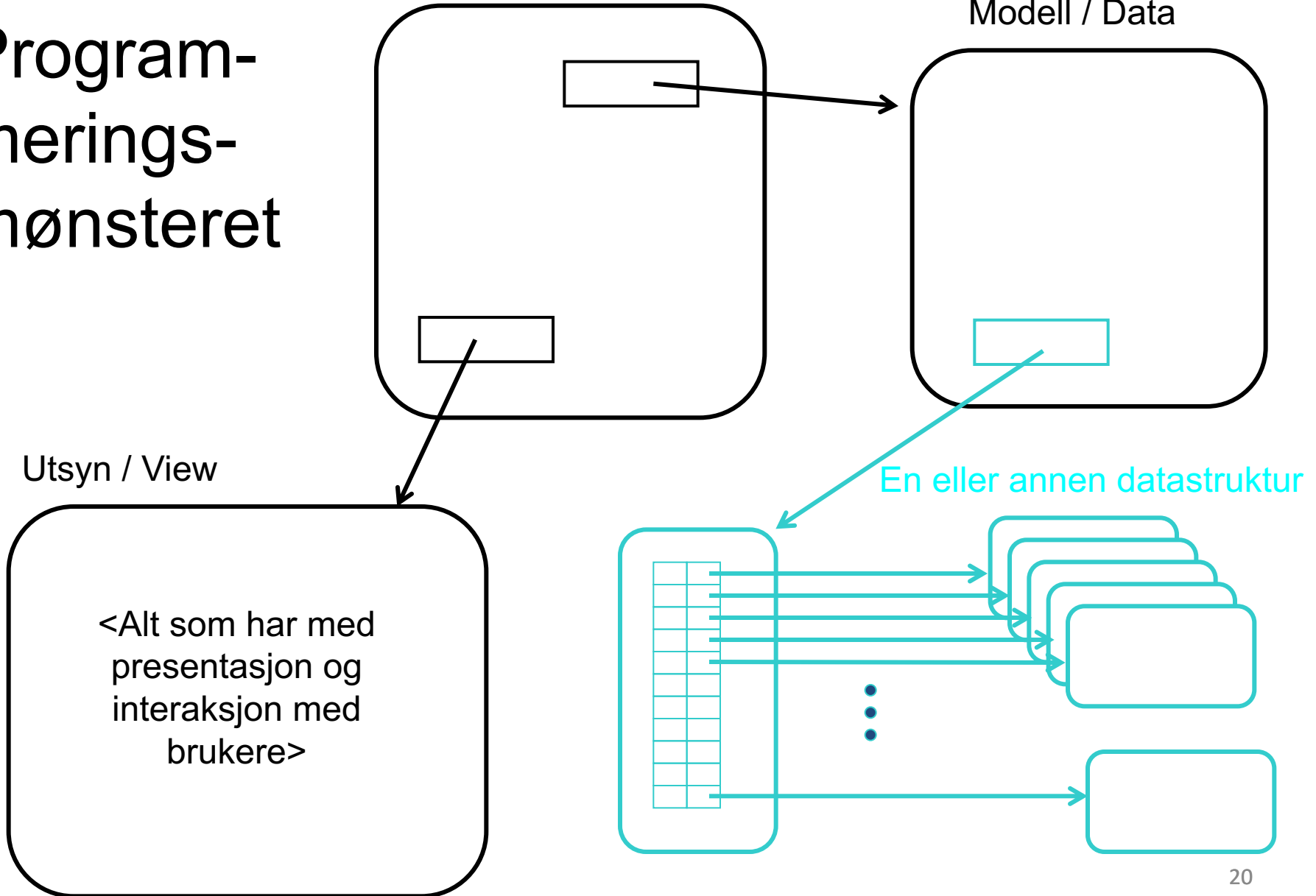
Kontroll

Modell / Data

Utsyn / View

<Alt som har med
presentasjon og
interaksjon med
brukere>

En eller annen datastruktur



Programskisse av
hele Banksystemet

```
/** Kontroll av Banksystemet */  
public class Bank {  
    private BankData b;  
    private BankUtsyn u;  
  
    public Bank( ) {  
        b = new BankData( );  
        u = new BankUtsyn( );  
    }  
  
    public static void main (String [] args) {  
        Bank bnk;  
        bnk = new Bank( );  
        bnk.ordreløkke( );  
    }  
  
    void ordreløkke ( ) {  
    }  
} // end Bank
```

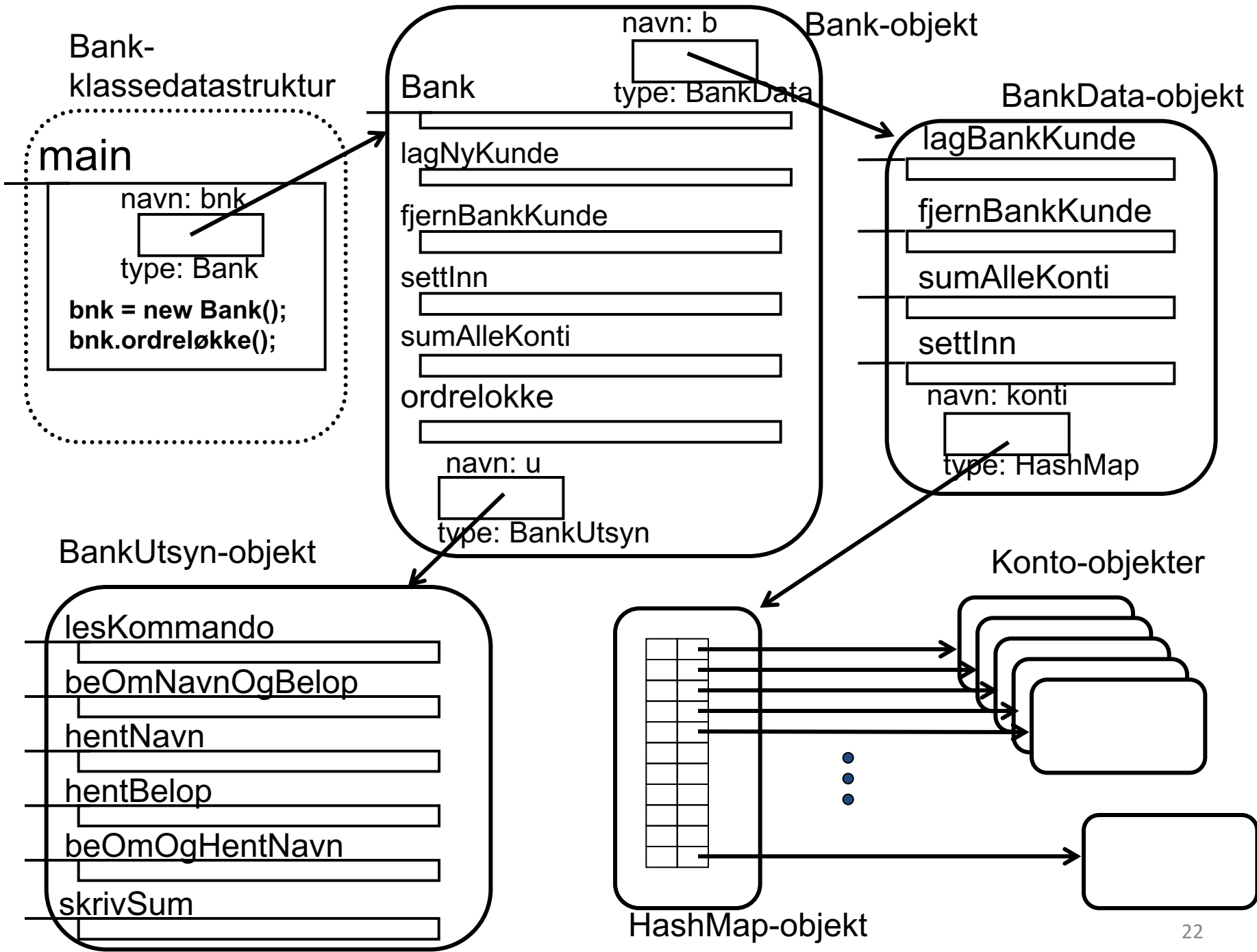
```
/** Utsyn for Bank*/  
class BankUtsyn {  
    <Alt som har med  
    presentasjon og  
    interaksjon med  
    brukere>  
  
}
```

```
/** BankData */  
class BankData {  
    <administrerer alle  
    kontoene>  
  
}
```

```
/** Konto */  
class Konto  
    <data om  
    en konto>  
  
}
```

Prøv å compilere (oversette)
og kjøre denne skissen





Fullstendig program

```
import java.util.*;
```

```
/**  
 * En konto har en eier identifisert av en tekst (String)  
 *  
 * @author Stein Gjessing  
 * @version 13. januar 2011  
 **/
```

```
class Konto {
```

```
    private String navn;  
    private double saldo;
```

```
/**  
 * Konstruktør setter kontoens saldo til 0.  
 *  
 * @param n    navn på kontoens eier  
 */
```

```
Konto (String n) {  
    navn = n;  
    saldo =0;  
}
```

```
/**  
 * Setter inn et beløp.  
 *  
 * @param b    beløp  
 */  
public void setInn(double b) {  
    saldo += b;  
}
```

```
/**  
 * Henter saldoen.  
 *  
 * @return    saldo på konto  
 */  
public double hentSaldo() {  
    return saldo;  
}
```

```
/**  
 * Henter eier.  
 *  
 * @return    navn på eier  
 */  
public String hentEier() {  
    return navn;  
}  
}
```



```
/**
 * BankData er modellen / dataene i banksystemet.
 *
 * Denne klassen innehold er alle metoder som
 * er nødvendige for å manipulere kontoer.
 * En ny bank inneholder ingen kontoer.
 * Alle kontoenes eiere må ha forskjellige navn
 *
 * @author Stein Gjessing
 * @versjon 13. januar 2011
 */
class BankData {

    private HashMap <String,Konto> konti =
        new HashMap<String,Konto>();

    /**
     * Oppretter en ny konto i banken.
     *
     * @param kunde  navn på kunden
     */
    public void lagBankKunde(String navn) {
        konti.put(navn, new Konto (navn));
    }

    /**
     * Fjerne en konto fra banken.
     *
     * @param navn  navn på kunde
     */
    public void fjernBankKunde(String navn) {
        konti.remove(navn);
    }
}
```

```
/**
 * Summerer saldoen for alle kontoene i banken.
 *
 * @return  summen av saldo for alle kontoer
 */
public double sumAlleKonti() {
    double sum = 0;

    for (Konto s: konti.values())
        sum+= s.hentSaldo();
    return sum;
}

/**
 * Setter inn penger på en konto.
 *
 * @param navn  navn på kunde
 * @param belop  beløpet som settes inn.
 */
public void settInn (String navn, double belop) {
    Konto k = konti.get(navn);
    k.setInn(belop);
}
}
```



```
/**
 * Bank er kontrollklassen for dette banksystemet.
 * Her ligger ordreløkken som styrer det hele.
 * Denne klassen er bindeleddet mellom
 * utsynet og datamodellen.
 *
 * @author Stein Gjessing
 * @version 17. Januar 2011
 */
public class Bank {
    private BankData b; private BankUtsyn u;
    Bank() { b = new BankData();
            u = new BankUtsyn();
    }

    public static void main(String [] args) {
        Bank bnk = new Bank();
        bnk.ordreløkke();
    }

    /**
     * Lager en ny konto.
     */
    void lagNyKunde () {
        String nvn = u.beOmOgHentNavn();
        b.lagBankKunde(nvn);
    }

    /**
     * Fjerner en konto.
     */
    void fjernBankKunde () {
        String nvn = u.beOmOgHentNavn();
        b.fjernBankKunde(nvn);
    }
}
```

```
/**
 * Setter inn penger på en konto.
 * Nødvendig informasjon hentes via utsynet.
 */
void settInn() {
    u.beOmNavnOgBelop(); String navn = u.hentNavn();
    double bel = u.hentBelop(); b.settInn(navn,bel);
}
/**
 * Henter summen av saldo fra alle kontoer
 * og viser resultatet ved hjelp av utsyn
 */
void sumAlleKonti() {
    double sum = 0; sum = b.sumAlleKonti();
    u.skrivSum(sum);
}
/**
 * Ordreløkken som styrer kommandoene.
 */
void ordreløkke () {
    int valg;
    valg = u.lesKommando();
    while(valg != 0) {
        switch (valg) {
            case 1: lagNyKunde(); break;
            case 2: fjernBankKunde(); break;
            case 3: settInn(); break;
            case 4: sumAlleKonti(); break;
            default: u.skrivFeil("Gi tall mellom 0-4");
        }
        valg = u.lesKommando();
    }
} // end ordreløkke
} // end class Bank
```



```

* BankUtsyn brukes til innlesing og visning av kundedata.
*
* @author Stein Gjessing
* @version 13. Januar 2011
*/
class BankUtsyn {
    private Scanner tast;
    // tast gir kortvarig oppbevaring av leste data
    private String navn;
    private double belop;

    /**
     * Konstruktør
     */
    BankUtsyn( ) {
        tast = new Scanner(System.in);
    }

    /**
     * Skriver ut menyen og leser inn kommandovalg.
     *
     * @return kommandovalg
     */
    public int lesKommando() {
        System.out.println("\nMeny: ");
        System.out.println("0 - avslutt");
        System.out.println("1 - Opprett ny kunde ");
        System.out.println("2 - Fjern kunde");
        System.out.println("3 - Sett inn penger");
        System.out.println("4 - Finn forvaltningskapital");
        System.out.print(" Velg funksjon: ");
        return (tast.nextInt());
    }
}

```

linje): ");

```

/**
 * Ber om navn og beløp og lagrer.
 */
public void beOmNavnOgBelop() {
    System.out.print("\nGi navn og beløp (på hver sin

    navn = tast.next();
    belop = tast.nextDouble();
}

/**
 * Henter navnet som er lest inn.
 *
 * @return navnet
 */
public String hentNavn() {
    return navn;
}

/**
 * Henter bløpet som er lest inn.
 *
 * @return beløpet
 */
public double hentBelop() {
    return belop;
}

/**
 * Ber om og henter et navn.
 *
 * @return navnet
 */
public String beOmOgHentNavn() {
    System.out.print("\nGi navn : ");
    return tast.next();
}

```



```
/**
 * Skriver ut bankens forvaltningskapital.
 *
 * @param sum kapitalen som skrives ut
 */
public void skrivSum(double sum) {
    System.out.println("\nBankens forvaltningskapital: "
        + sum);
}

/**
 * Skriver ut en feilmelding.
 *
 * @param feil feilmeldingen
 */
public void skrivFeil(String feil) {
    System.out.println(feil);
}

} // end BankUtsyn
```

Lag Javadoc:

```
>javadoc -package Bank.java
```

Design - kjente grep for å strukturere koden

- Top-down nedbryting
 - av moduler til flere enklere moduler i ett eller flere nivåer
 - Objekter kan nyttes som moduler. Top-down nyttes for noen nivåer.
- Bottom-up programmering
 - Lage generelle klasser og operasjoner (metoder) vi vil trenge i systemet vårt fra "bunnen av" eller med grunnlag i biblioteker
 - jfr. Java-biblioteket med mer enn 4000 klasser for de fleste behov.



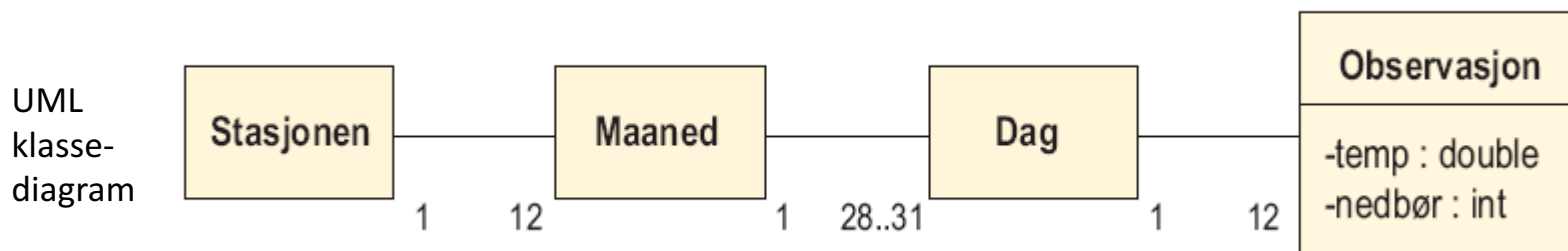
Hvor skal vi legge metodene (funksjonene)

- Kan synes opplagt, men er det ikke alltid
 - To prinsipper (gir ofte, men ikke alltid samme resultat):
 - Legg en funksjon der (nesten alle) data er som funksjonen skal jobbe på
 - Legg funksjonene der de hører hjemme naturlig / i virkeligheten
 - Ofte samme resultatet, men ikke her:
 - Oppgave:
 - I et student/eksamens-system skal det skrives ut vitnemål.
Skal metoden skrivVitemål() legges i:
 - Class Student - hvor data ligger
 - Class SkoleAdministrasjon - hvor funksjonen utføres i den virkelige verden
- Velg det siste – da blir det lettere å vedlikeholde systemet
(SkoleAdministrasjon henter data fra Student)

Top-down nedbrytning av en beregning

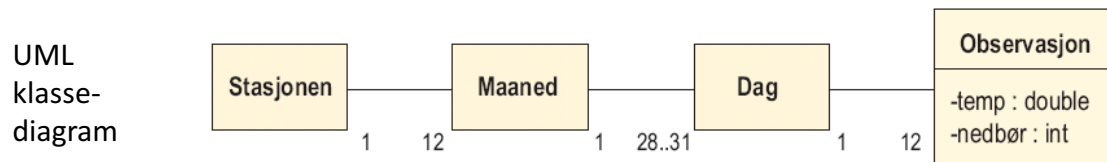
Eksempel værdata fra Met. Inst. :

Anta at vi har en enkel modell av værobservasjoner for Meteorologisk institutt, og at observasjonene gjøres på værstasjoner. Annenhver time, hver dag, hele året igjennom registreres temperatur og nedbør siden forrige observasjon. Nå kan vi tenke oss at vi er interessert i maksimums- og minimumstemperatur hver dag, og i gjennomsnittlig temperatur og samlet nedbør hver dag, hver måned og for hele året fra disse observasjonene.



Hvordan skal vi regne ut gj.snitt nedbør for en stasjon per år?

Top-down: Gj.snitt nedbør per år og mnd.



- Enten en metode i class Stasjon med tre løkker inne i hverandre, over alle Måneder, Dager, Observasjoner
 - Komplikasjon: Håndtering av manglende data på alle nivåer
- Alternativt - bedre: Hver av klassene har sin beregnGjennomsnittNedbør() – metode, som bare kaller neste tilsvarende det riktige antall ganger (og som til sist leser av verdien i Observasjon)



Nytt eksempel:

Tilbake til 9. februar og Interface

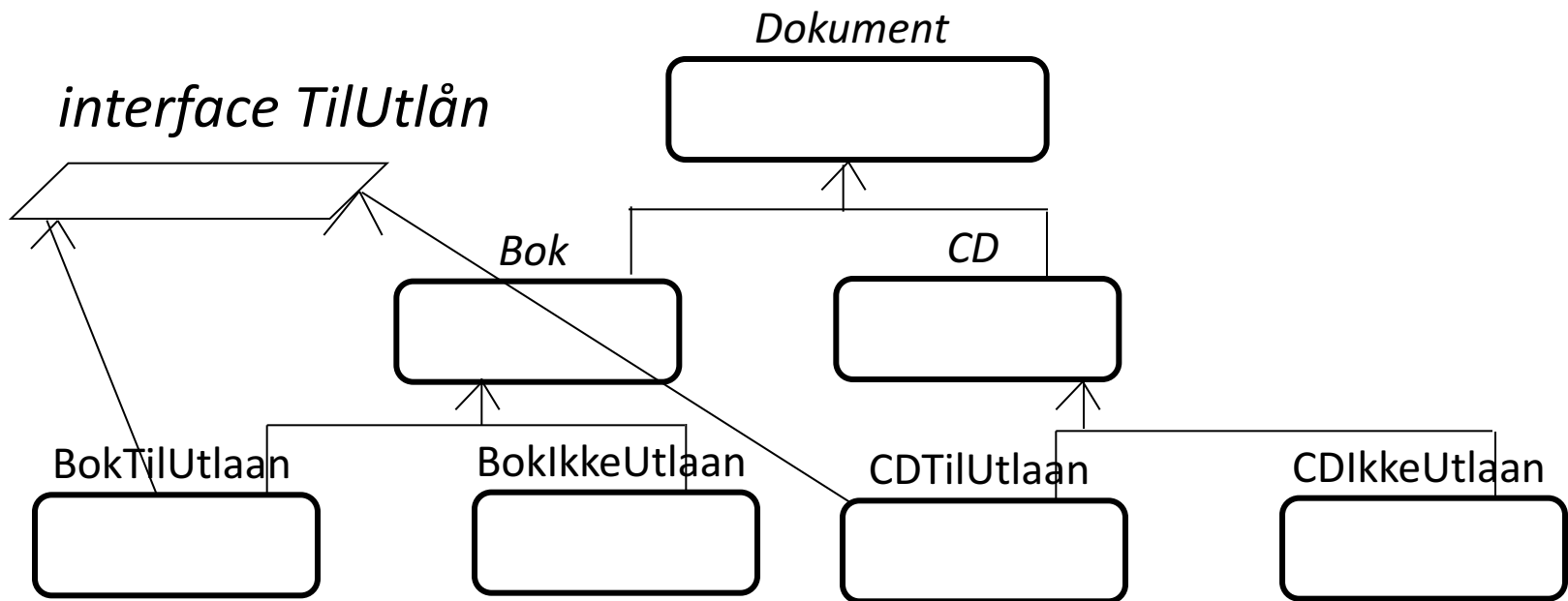
Oppgave:

Lag et program som administrerer et bibliotek.

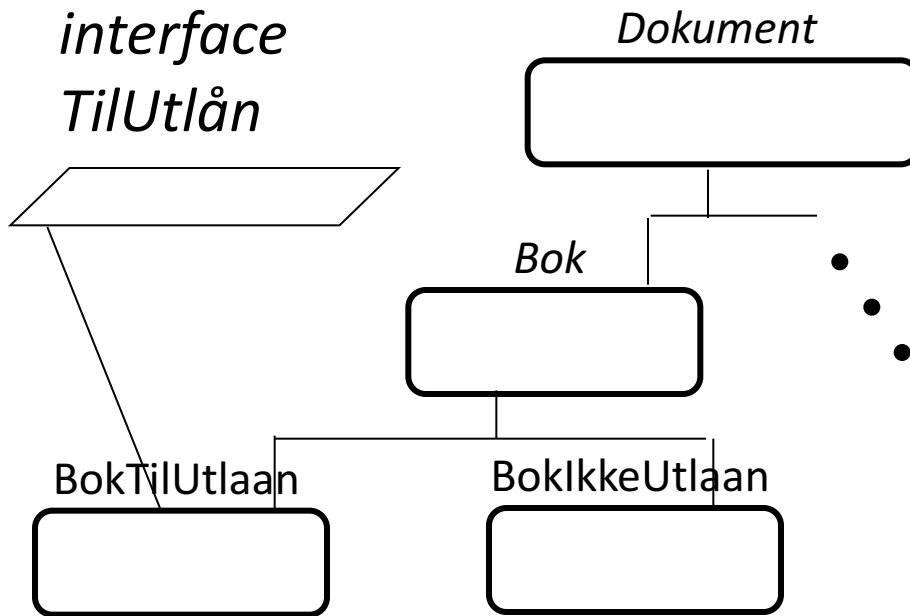
Analyse av et bibliotek:

- Bøker, tidsskrifter, CDer, videoer, mikrofilmet materiale, antikvariske bøker, flerbindsverk, oppslagsverk, upubliserte skrifter, ...
- En del felles egenskaper
 - antall eksemplarer, hylleplass, identifikasjonskode (Dokument)
 - for det som kan lånes ut: Er utlånt ? , navn på låner, ... (TilUtlån)
 - for det som er antikvarisk: Verdi, forskringssum, ... (Antikvarisk)
- Spesielle egenskaper:
 - Bok: Forfatter, tittel, forlag
 - Tidsskriftnummer: Årgang, nummer, utgiver
 - CD: Tittel, artist, komponist, musikkforlag

Klassehierarki, forenklet bibliotek



Forenklet bibliotek



```
abstract class Dokument {  
    String tittel;  
}
```

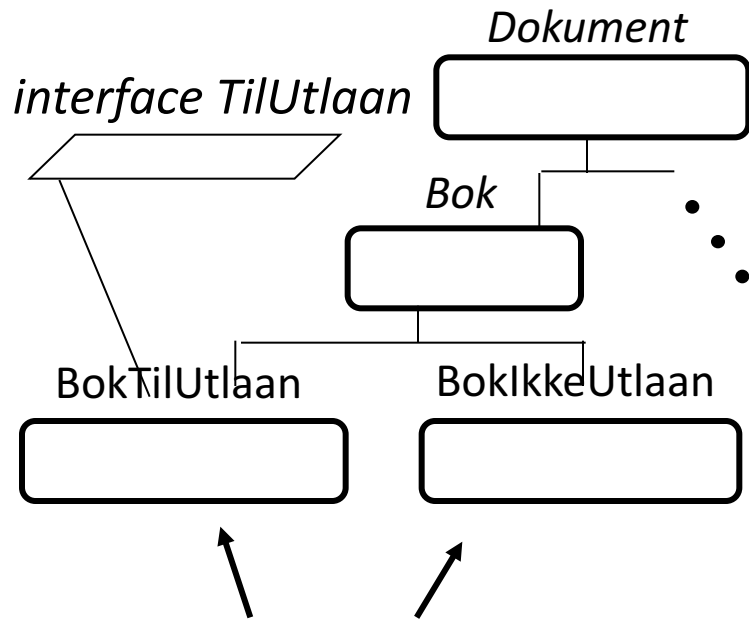
```
abstract class Bok extends Dokument {  
    String forlag;  
    int trykningsår;  
}
```

```
interface TilUtlaan {  
    abstract void låne(String låner) ;  
    abstract void levere() ;  
    abstract boolean utlånt() ;  
    static final String ingen = "ingen";  
} // Slutt interface TilUtlaan
```



"abstract" betyr at vi ikke kan lage slike objekter

BokTilUtlaan



Bare disse to klassene kan vi lage objekter av

BokTilUtlaan har både egenskapene til Bok og egenskapene til TilUtlaan. Objekter av denne klassen kan spille begge rollene!

```
class BokTilUtlaan extends Bok
    implements TilUtlaan {
    String låner = ingen;

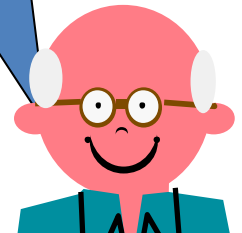
    public void låne (String l) {
        låner = l;
    }
    public void levere() {
        låner = ingen;
    }
    public boolean utlånt() {
        return låner != ingen;
    }
} // Slutt class BokTilUtlaan
```

```
class BokIkkeUtlaan extends Bok {
}
```

Se på implementasjonen igjen:

```
interface TilUtlaan {  
    abstract void låne(String låner) ;  
    abstract void levere() ;  
    abstract boolean utlånt() ;  
    static final String ingen = "ingen";  
}
```

Metodene i et
interface er
veldig polymorfe



```
class BokTilUtlaan extends Bok implements TilUtlaan {  
    String låner = ingen;  
    public void låne (String l) {  
        låner = l;  
    }  
    public void levere() {  
        låner = ingen;  
    }  
    public boolean utlånt() {  
        return låner != ingen;  
    }  
}
```

Dette er de tre
metodene som vi
må love å
implementere

```
interface TilUtlaan {  
    abstract void låne(String låner) ;  
    abstract void levere() ;  
    abstract boolean utlånt() ;  
    static final String ingen = "ingen";  
}
```

interface TilUtlaan

Dokument



CD



CDTilUtlaan



CDIkkeUtlaan



```
abstract class CD extends Dokument {  
    String komponist, artist, musikkforlag;  
}
```

```
class CDTilUtlaan extends CD implements TilUtlaan {  
    String låner = ingen;
```

```
    public void låne(String l) { låner = l; }
```

```
    public void levere() { låner = ingen; }
```

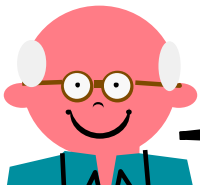
```
    public boolean utlånt() { return låner != ingen; }
```

```
} // Slutt class CDTilUtlaan
```

**Her er
de tre
metodene
igjen**

Husk om interface:

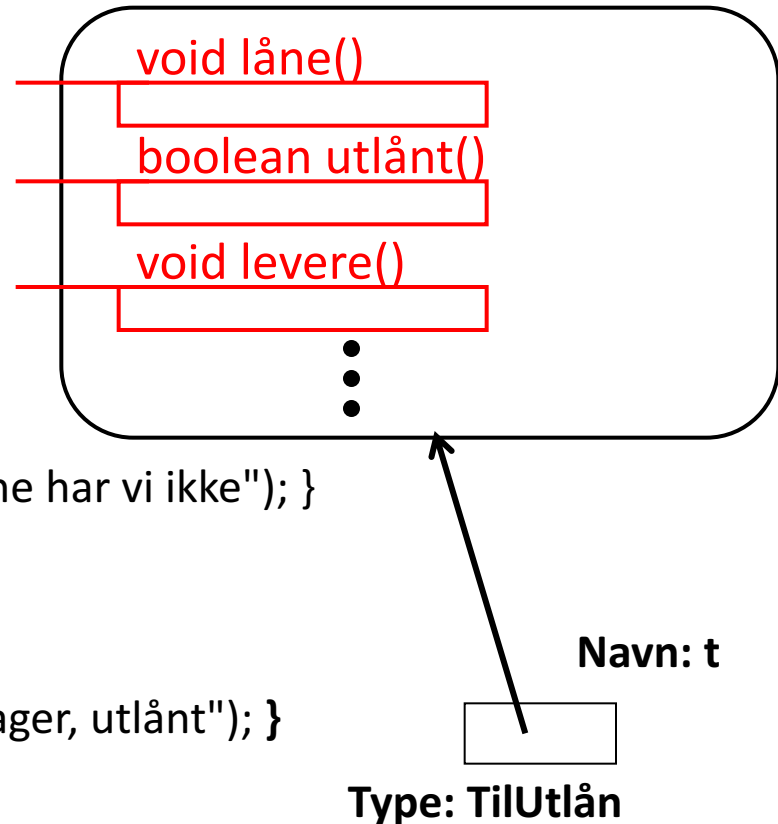
- Navnet på et interface kan brukes som typenavn når vi lager pekere.
- Vitsen med et grensesnitt er å spesifisere **hva** som skal gjøres (ikke hvordan)
- Ofte er det flere implementasjoner av et grensesnitt (flere klasser kan implementere det).
- En implementasjon (av et grensesnitt) skal kunne endres uten at resten av programmet behøver å endres.



Men selv om et grensesnitt skal spesifisere hva (og ikke hvordan), spesifiseres bare syntaksen (signaturen) og ikke hva som gjøres (semantikken)

Metoden låne

```
void låne() throws IOException {  
    Dokument d = null;  
    String h;  
    System.out.println("Tittel: ");  
    h = <les tittel>;  
    d = alleDokumenter.get(h);  
    if (d==null) {System.out.println("Beklager, denne har vi ikke"); }  
    else  
        if (d instanceof TilUtlaan) {  
            TilUtlaan t = (TilUtlaan) d;  
            if (t.utlånt()) { System.out.println("Beklager, utlånt"); }  
            else {  
                System.out.print("Låners navn: ");  
                String n = <les navn>;  
                t.låne(n);  
            }  
        }  
    } else { System.out.println("Beklager, denne låner vi ikke ut");}  
}
```



Metoden levereTilbake

```
void levereTilbake() {  
    Dokument d = null;  
    String h;  
    System.out.print("Tittel: ");  
    h = <les tittel>;  
    d = alleDokumenter.get(h);  
    if (d==null)  
        {System.out.println("Beklager, feil tittel"); }  
    else  
    if (d instanceof TilUtlaan) {  
        TilUtlaan t = (TilUtlaan) d;  
        if (t.utlånt()) {  
            t.levere();  
            System.out.println("Takk");  
        }  
    else {System.out.println("Beklager, denne er ikke utlånt");}  
    }  
}
```

