Videregående programmering i Java

Dag 4 - objektorienteret design

& hyppigt anvendte designmønstre

Hyppigt anvendte designmønstre: Proxy, Adapter, Iterator, Facade, Dynamisk Binding
Objektorienteret design
Udskudt JNI og kald til maskinkode/C/C++ fra Java
Udgår Styring af eksterne komponenter (OpenOffice.org / COM)
Udskudt Automatiseret afprøvning med JUnit

Modularitet

- Programmer er ofte for store og for komplekse til at blive designet og implementeret som ét system.
- De bør derfor deles op i håndtérbare moduler
  - "Del og hersk"
- Principper for modulerne:
  - Hvert modul bør have et klart ansvarsområde
  - Modulerne bør have lav kobling indbyrdes

Kohæsion og kobling

- Kohæsion
  - Hvor relaterede funktionerne, der varetages inden for et modul, er
  - Metoderne i en klasse bør gøre relaterede ting
  - Ergo: Høj kohæsion er godt
- Kobling
  - Handler om graden af afhængighed, der er mellem modulerne
  - Modulerne bør stort set være uafhængige og kun løst forbundet
  - Ergo: Lav kobling er godt

Ansvarsområder

Ansvarsområder for et objekt/modul kan være:

- At oprette nye objekter eller udføre en beregning
- At foretage en handling i andre objekter
- At kontrollere og koordinere aktiviteter i andre objekter
- At kende private data
- At kende relaterede objekter
- At kende ting, som objektet kan beregne

- Høj kohæsion (sammenhæng - eng.: high cohesion)
  - at et objekt har ét overskueligt og let forståeligt ansvarsområde (eller eventuelt flere områder tæt relaterede til hinanden)

Designfase

Hvordan skal programmet fungere

Redskaber til objektorienteret design

- Ord til klasser
- Kollaborationsdiagrammer
- Klassediagrammer

Vi bruger nu 10 minutter til hvert punkt!

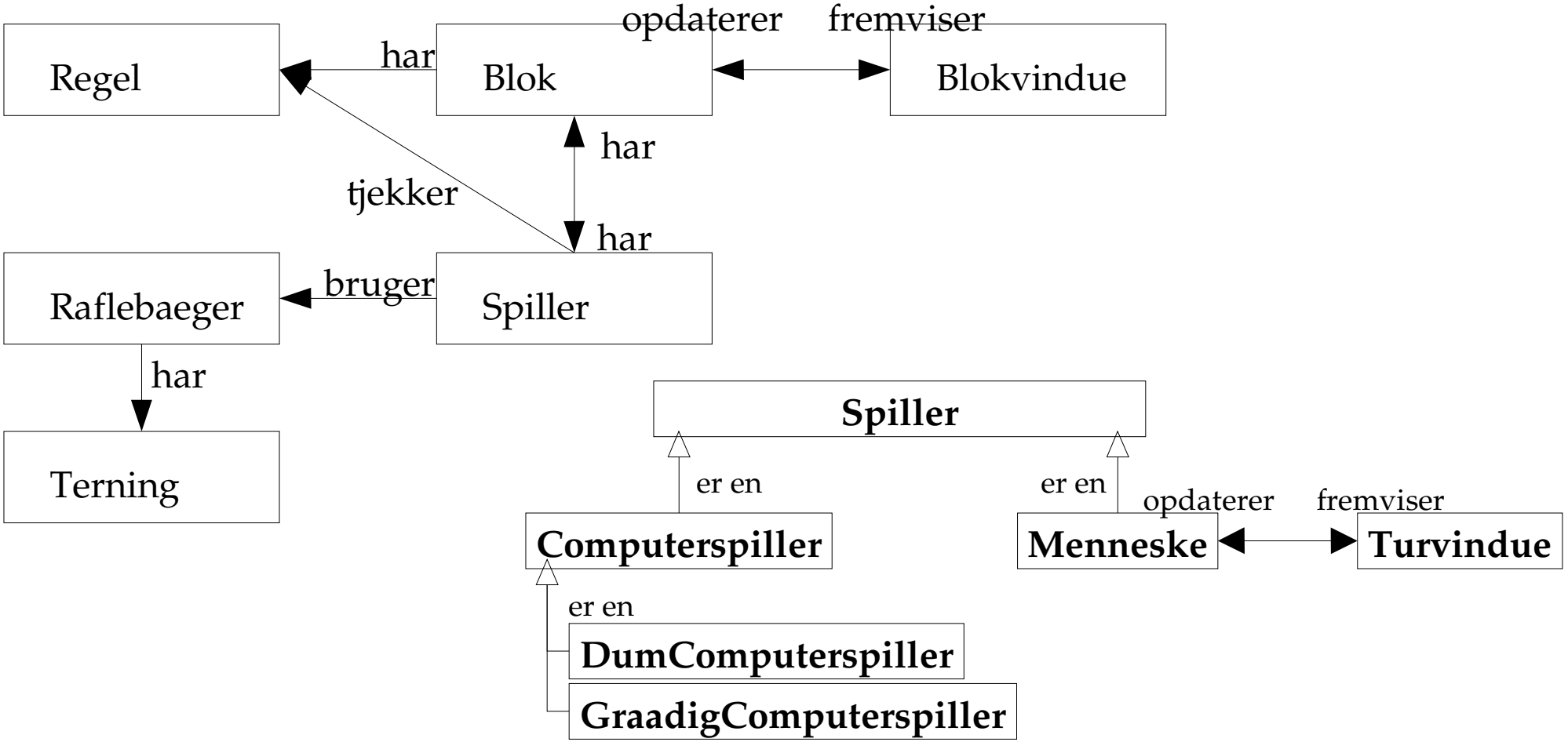
Fremlæggelse næste gang!

Ord til klasser

- Tommelfingerregler:
  - Navneord (substantiver) i ental bliver ofte til klasser
  - Klassenavne skal altid være i ental
  - Udsagnsord (verber) bliver ofte til metoder

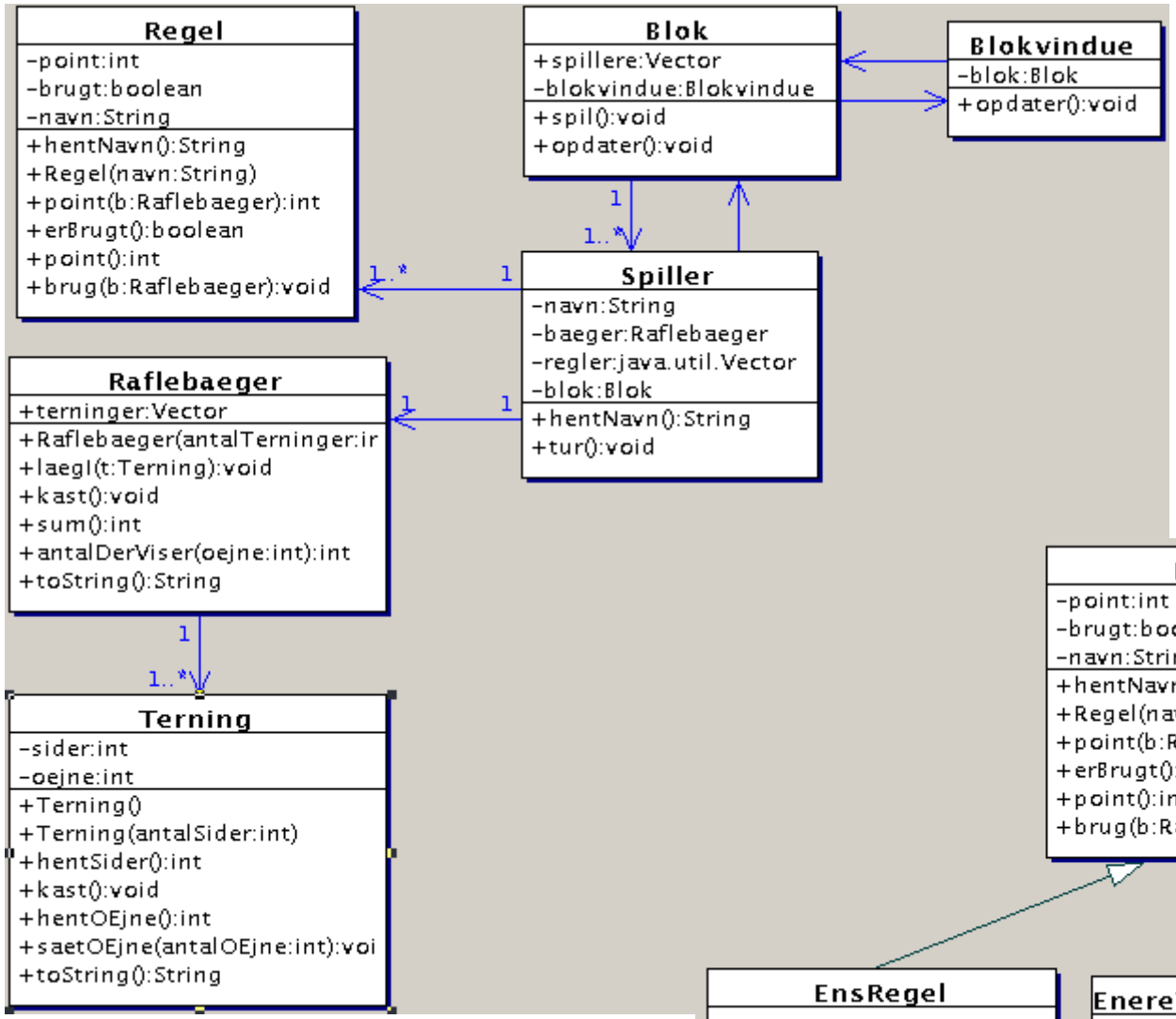
Kollaborationsdiagrammer

- Husk ansvarsområder
- Forskelligt antal => forskellige klasser
- Tegn (har)relationer på



Klassediagrammer

- To-vejs-redigering med et UML-udviklingsværktøj
  - TogetherJ
  - JDeveloper
  - PoseidonUML



Designmønstre

http://www.javacamp.org/designPattern/

http://www.fluffycat.com/java/patterns.html

(http://www.jguru.com/faq/Patterns)

Designmønstret Adapter

Problem: Et system forventer et objekt af en bestemt type (der implementerer et bestemt interface eller arver fra en bestemt klasse), men det objekt, man ønsker at give til systemet, har *ikke* denne type.

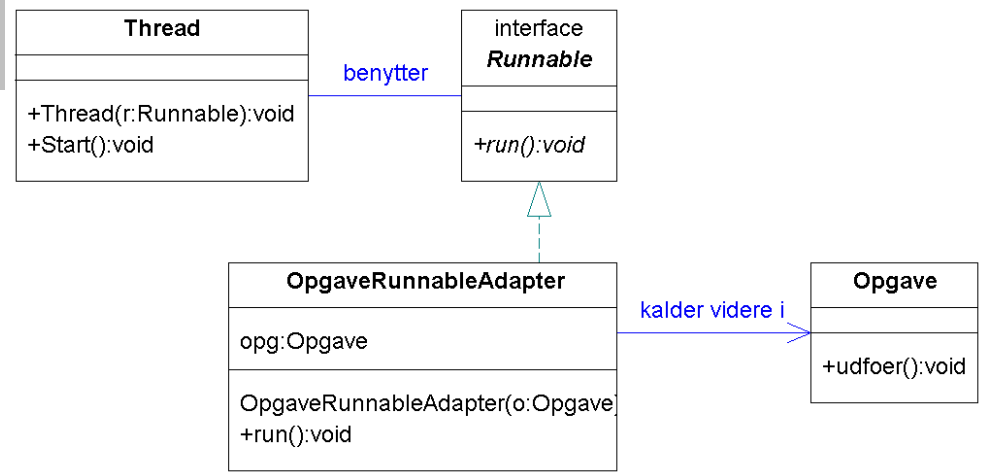
Løsning: Definér et Adapter-objekt af den type, som systemet forventer, og lad Adapter-objektet delegere kaldene videre til det rigtige objekt.

- En Adapter fungerer som omformer mellem nogle klasser
- Få et objekt til at passe ind i et system ved at bruge et Adapter-objekt, der passer ind i systemet, og som kalder videre i det rigtige objekt

Designmønstret Adapter

- Eksempel: Få et Opgave-objekt omformet til at passe til Threads forventning om et Runnable-objekt

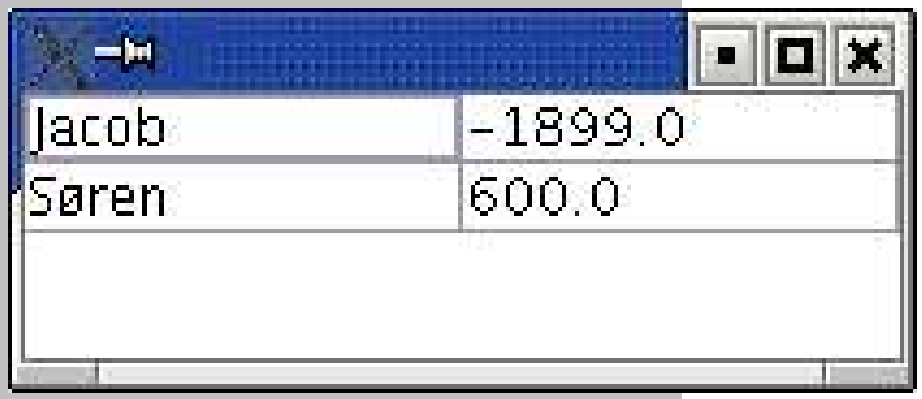
```
public class OpgaveRunnableAdapter implements Runnable
{
    Opgave opg;
    OpgaveRunnableAdapter(Opgave o) { opg = o; }
    public void run() {
       // Oversæt kald af run() til kald af udfør()
       opg.udfør();
    }
}
```



Designmønstret Adapter

- Eksempel: Få data (en liste af Kunde-objekter) til at passe ind i en JTables forventning om et TableModel-objekt

```
import java.util.*;
import javax.swing.table.*;

public class KundelisteTableModelAdapter extends AbstractTableModel
{
    private List liste;

    public KundelisteTableModelAdapter(List liste1) { liste = liste1; }

  public int getRowCount() { return liste.size(); }

  public int getColumnCount() { return 2; }  // navn og kredit

  public String getColumnName(int kol)
    {
      return kol==0 ? "Navn" : "Kredit";
    }

  public Object getValueAt(int række, int kol)
    {
      Kunde k = (Kunde) liste.get(række);
      return kol==0 ? k.navn : ""+k.kredit;
    }
}
```



Designmønstret Proxy

Problem: Et objekt, der bliver brugt af klienten, skal nogen gange bruges lidt anderledes, men ikke altid, så det er uhensigtsmæssigt at ændre i klassen eller i klienten.

Løsning: Lav en Proxy-klasse, der *lader* som om, den er det rigtige objekt, og kalder videre i det rigtige objekt.

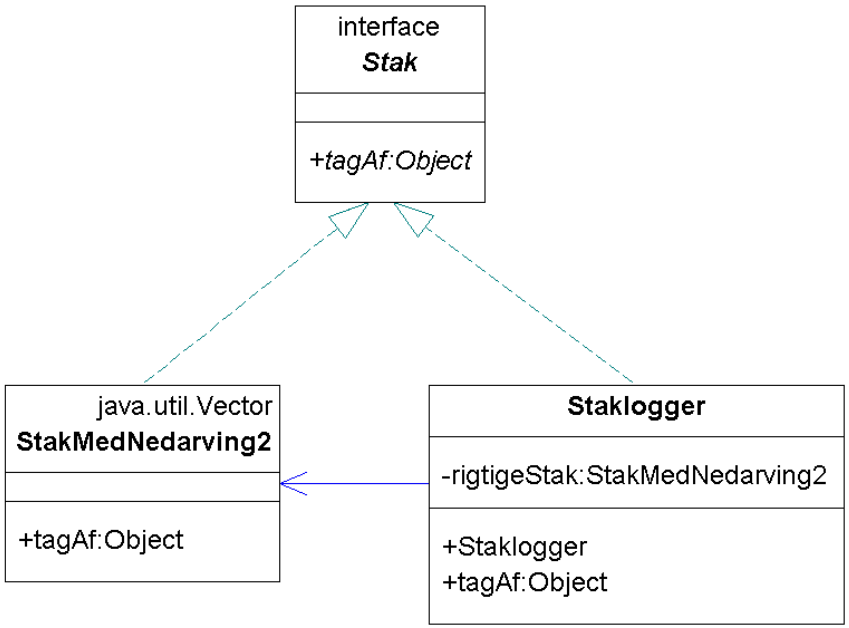
- Proxy på dansk "stråmand" eller "mellemmand"
  - Ordet brugtes oprindeligt i banksektoren
  - Internet: En proxy-server
- Oftest ved klienten ikke at den bruger en proxy. Når proxyen bliver kaldt, vil den som regel delegere kaldet videre til det andet objekt, men den kan også vælge f.eks.:
  - at returnere med det samme og udføre kaldet i baggrunden
  - at afvise kaldet (f.eks. ved at kaste en undtagelse)
  - at udføre kaldet på en anden måde (f.eks. anderledes parametre)

Designmønstret Proxy

- Staklogger: en Stak, der delegerer videre til en anden Stak:

```
Stak s = new Staklogger( new StakMedNedarving2() );
```

```
public class Staklogger implements Stak
{
   private Stak rigtigeStak;

   public Staklogger(Stak s) {  rigtigeStak = s; }

   public void lægPå(Object o)
   {
      System.out.print("Staklogger: lægPå("+o+")");
      rigtigeStak.lægPå(o);
   }

   public Object tagAf()
   {
      Object o = rigtigeStak.tagAf();
      System.out.print("Staklogger: tagAf() gav: "+o);
      return o;
   }
}
```



Designmønstret Proxy

- Uforanderlige samlinger:

```
Collection d = new ArrayList();
d.add("Hej");
d.add("med");
d.add("dig");

d = new UforanderligSamling(d);              // Her: vores egen implementation
// d = Collections.unmodifiableCollection(d) // Her: fra standardbiblioteket
// herefter kan dataene ikke mere ændres gennem d
...
d.add("igen"); // undtagelse opstår
```

- I standardbiblioteket findes
  - Collections.unmodifiableCollection(): Uforanderligt Proxy-objekt
  - Collections.synchronizedCollection(): Trådsikkert Proxy-objekt


```
public class UforanderligSamling implements Collection, Serializable
{
   private Collection c; // til videredelegering

   UforanderligSamling(Collection c) {
     if (c==null) throw new NullPointerException();
     this.c = c;
   }

   // videredelegering af kald, der ikke ændrer samlingen c
   public int size()                          { return c.size(); }
   public boolean isEmpty()                   { return c.isEmpty(); }
   public boolean contains(Object o)          { return c.contains(o); }
   public boolean containsAll(Collection coll) { return c.containsAll(coll); }
   public String toString()                   { return c.toString(); }

   private static void fejl() {
     throw new UnsupportedOperationException("Denne samling kan ikke ændres");
   }

   // afvisning af kald, der ændrer samlingen
   public void clear()                    { fejl(); }
   public boolean add(Object o)           { fejl(); return false; }
   public boolean remove(Object o)        { fejl(); return false; }

   // iteratorer skal afvise ændringer, men ellers fungere som c's iterator
   public Iterator iterator()
   {
     return new Iterator() {       // anonym klasse, der implementerer Iterator
       Iterator i = c.iterator();  // videredelegering til c's iterator
       public boolean hasNext() { return i.hasNext(); }
       public Object next()     { return i.next(); }
       public void remove()     { fejl(); }
     };
   }
}
```

Variationer af designmønstret Proxy

- Fjernproxy (eng.: Remote Proxy) - bruges, når man har brug for en lokal repræsentation af et objekt, der ligger på en anden maskine. Afsnit 16.8.2 Dataforbindelse over netværk er et eksempel på dette. RMI (Remote Method Invocation) beskrevet i afsnit 14.3 anvender også dette princip.
- Cache - fungerer som proxy for et objekt med nogle omkostningsfulde metodekald. I de tilfælde hvor en tidligere cachet returværdi fra metodekaldet kan bruges, foretages kaldet ikke, men den cachede værdi returneres i stedet (se afsnit 16.8.3, Dataforbindelse, der cacher forespørgsler).
- Adgangssproxy - bestemmer, hvad klienten kan gøre med det virkelige objekt (Eksempel: UforanderligSamling).
- Virtuel Proxy - udskyder oprettelsen af omkostningsfulde objekter, indtil der er brug for dem.

Doven Initialisering/Virtuel Proxy

```
public class VirtuelStak implements Stak
{
    private Stak rigtigeStak;

    public void lægPå(Object o)
    {
       if (rigtigeStak==null) rigtigeStak = new StakMedNedarving2();
       rigtigeStak.lægPå(o);
    }

    public Object tagAf()
    {
       if (rigtigeStak==null) rigtigeStak = new StakMedNedarving2();
       return rigtigeStak.tagAf();
    }
}
```

- Fordele og ulemper:
  - Det rigtige objekt kan ikke oprettes endnu, f.eks. fordi det afhænger af andre objekter, der ikke er klar endnu,
  - At oprette det rigtige objekt er dyrt i hukommelses- eller CPU-forbrug, og det er måske slet ikke sikkert, at programmet kommer til at bruge objektet, så det er en fordel at udskyde oprettelsen.
  - Hver gang objektet skal bruges, skal det først tjekkes, om det rigtige objekt er blevet oprettet.

Designmønstret Iterator

Problem: Du er i gang med at lave et system, som andre (klienter) skal anvende, hvor de skal kunne gennemløbe dine data. Du ønsker ikke, at de skal kende noget til, hvordan data er repræsenteret i dit system (f.eks. antal elementer eller deres placering i forhold til hinanden).

Løsning: Definér et hjælpeobjekt (en Iterator), som klienten kan bruge til at gennemløbe data i dit system.

En Iterator er beregnet til at gennemløbe data

- En Iterator har som minimum:
  - en metode til at spørge, om der er flere elementer, og
  - en metode til at hente næste element
- En Iterator bruges i stedet for en tællevariabel. Fordelen ved at definere en Iterator er, at klienten *ikke behøver at vide noget om strukturen af de data, der gennemløbes.*

```
public class UforanderligSamling implements Collection, Serializable
{
   private Collection c; // til videredelegering

   UforanderligSamling(Collection c) {
     if (c==null) throw new NullPointerException();
     this.c = c;
   }

   // videredelegering af kald, der ikke ændrer samlingen c
   public int size()                          { return c.size(); }
   public boolean isEmpty()                   { return c.isEmpty(); }
   public boolean contains(Object o)          { return c.contains(o); }
   public boolean containsAll(Collection coll) { return c.containsAll(coll); }
   public String toString()                   { return c.toString(); }

   private static void fejl() {
     throw new UnsupportedOperationException("Denne samling kan ikke ændres");
   }

   // afvisning af kald, der ændrer samlingen
   public void clear()                   { fejl(); }
   public boolean add(Object o)          { fejl(); return false; }
   public boolean remove(Object o)       { fejl(); return false; }

   // iteratorer skal afvise ændringer, men ellers fungere som c's iterator
   public Iterator iterator()
   {
     return new Iterator() {       // anonym klasse, der implementerer Iterator
       Iterator i = c.iterator();  // videredelegering til c's iterator
       public boolean hasNext() { return i.hasNext(); }
       public Object next()     { return i.next(); }
       public void remove()     { fejl(); }
     };
   }
}
```

Designmønstret Facade

Problem: Et sæt af beslægtede objekter er indviklede at bruge, og der er brug for en simpel grænseflade til dem.

Løsning: Definér et hjælpeobjekt, en Facade, der gør objekterne lettere at bruge.

En Facade er altså et objekt, der giver en "brugergrænseflade" til nogle andre objekter og dermed forenkler brugen af disse objekter.

- Eksempel: URL
  - URL er facade til URLConnection
- Eksempel: Socket og ServerSocket
  - Er to forskellige facader til SocketImpl, der varetager den egentlige netværkskommunikation

Designmønstret Dynamisk Binding

Problem: Programmet skal senere kunne udvides til at bruge nogle flere klasser, uden at programmet skal skrives om.

Løsning: Definér et fælles interface (eller superklasse) for klasserne, og søg efter egnede klasser på køretidspunktet, indlæs dem og brug dem.

- Indlæs klasser dynamisk under kørslen, efter at programmet er startet.
- Dynamisk Binding/ Lænkning (eng.: Dynamic Linkage)
- Class.forName()

```
Class klassen = Class.forName("java.util.Vector");
Object objektet = klassen.newInstance();
```

  - Eksempel: JDBC

Designmønstret Dynamisk Binding

```
public interface Funktion
{
  public double beregn(double x);
}
```

```
public class Funktion_sin implements
{
  public double beregn(double x)
  {
    return Math.sin(x);
  }
}
```

```
public class BenytFunktionsfortolker
{
  public static void main(String arg
  {
    FunktionsfortolkerDynBind analysa
    Funktion f = analysator.fortolk('
    System.out.println("f(1)=" + f.be
  }
}
```

```
public class FunktionsfortolkerDynBind
{
  public Funktion findFunktion(String navn)
  {
    try
    {
      // Prøv at indlæse en klasse der hedder f.eks. Funkt
      Class klasse = Class.forName("Funktion_"+navn);

      // Opret et objekt fra klassen
      Funktion f = (Funktion) klasse.newInstance();
      return f;
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
      throw new IllegalArgumentException("ukendt funktion:
    }
  }

  public Funktion fortolk(String udtryk)
  {
    // endnu ikke implementeret - returner bare noget.
    return findFunktion("sin");
  }
}
```