Objektorienterede metoder

Denne gang: Designmønstre

Skabende designmønstre:
Fabrikeringsmetode/Fabrik, Singleton,
Abstrakt fabrik (Toolkit), Prototype, Objektpulje
Singleton-eksempel: Forskellige slags dataforbindelser

Hyppigt anvendte designmønstre: Proxy, Adapter, Iterator,
Facade, Dynamisk Binding

Næste gang: Automatiseret afprøvning med JUnit

Læsning: javabog.dk/VP kapitel 16, 17

Dette materiale er under Åben Dokumentlicens, se http://www.sslug.dk/linuxbog/licens.html

Designmønstre

- Navngiven måde at programmere på
- Beskrivelse af, hvordan et bestemt slags problem kan løses
  - Beskrivelse af konsekvenserne af denne måde at løse problemet på.
- Det samme designmønster kan løse et lignende problem i et andet program
  - Engelsk: (Reusable) Design Pattern
    - Mønster = noget, som gentages
    - Design = hvordan et program er sat sammen
      - Begreb fra OOAD – Objektorienteret analyse og design
      - Designet vises ofte som et klassediagram

Formål med designmønstre

- At give en fælles begrebsramme
  - Lettere at forklare hvad man har lavet
  - Klar begrebsramme lettere at forstå
  - Lettere at beslutte hvordan noget laves
- Ikke "opfinde den dybe tallerken" igen.
  - Velafprøvede idéer til godt design
  - Undgå de mest almindelige faldgruber
- Mindske graden af bindinger (kobling) mellem forskellige dele af et program

Eksempler på designmønstre

- Singleton
  - sikring af, at der kun eksisterer ét objekt af en bestemt slags
- Prototype
  - objekter oprettes ud fra eksisterende skabelon-objekter
- Objektpulje
  - genbrug de samme objekter igen og igen ved at huske dem i en pulje
- Adapter
  - får et objekt til at passe ind i et system ved at fungere som omformer mellem objektet og systemet
- Kommando
  - registrér brugerens handlinger, så at de kan fortrydes igen
- Observatør/Lytter
  - 'abonnere' på, at en ting (hændelse) sker

Designmønstre

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

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

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

Skabende designmønstre

```
// høj kobling - klient opretter et Hjælp-objekt
Hjælp h = new Hjælp();



...
h.metode1();
h.metode2();
```

MEN... det kunne være at:

- Det var en nedarving af Hjælp, der skulle oprettes (polymorfi)
- Objektet skulle oprettet med nogle bestemte parametre i konstruktøren
- Det samme objekt skulle bruges af alle klienter (en Singleton)
- Eksisterende Hjælp-objekter skulle genbruges (en Objektpulje)

Skabende designmønstre

```
// høj kobling - klient opretter et Hjælp-objekt
Hjælp h = new Hjælp();

...
h.metode1();
h.metode2();
```

MEN... det kunne være at:

- Det var en nedarving af Hjælp, der skulle oprettes (polymorfi)
- Objektet skulle oprettet med nogle bestemte parametre i konstruktøren
- Det samme objekt skulle bruges af alle klienter (en Singleton)
- Eksisterende Hjælp-objekter skulle genbruges (en Objektpulje)

- Oprettelse af objekt afgør objektets præcise type!
  - Dette er (for) stærk kobling i visse tilfælde
- Den del af programmet (klienten) som *bruger* visse objekter skal ikke altid også *oprette* disse objekter

Skabende designmønstre

```
// høj kobling - klient opretter et Hjælp-objekt
//Hjælp h = new Hjælp();

// fabrikeringsmetode leverer objekt til klienten
Hjælp h = Hjælp.opretHjælp();

...
h.metode1();
h.metode2();
```

Det kan være at:

- Det er en nedarving af Hjælp, der bliver oprettet (polymorfi)
- Objektet bliver oprettet med nogle bestemte parametre i konstruktøren
- Det samme objekt bliver brugt af alle klienter (en Singleton)
- De eksisterende Hjælp-objekt bliver genbrugt (en Objektpulje)

- Fabrikeringsmetode (eng.: Factory Method)
  - En metode, der opretter et objekt for klienten
  - Afkobler (mindsker graden af bindinger) mellem
    - oprettelsen af nogle bestemte objekter (i ét modul)
    - anvendelsen af dem af (i et andet modul, klienten)

Designmønstret Fabrik

(eng.: Factory)

(objekt med fabrikeringsmetode)

Problem: Klienten kan/skal ikke bestemme præcist, hvordan nogle objekter oprettes.

Løsning: Lad en Fabrik med en fabrikeringsmetode varetage oprettelsen.

```
// høj kobling - klient opretter et Hjælp-objekt
Image i = new Image("billede.gif");      // forkert!!
```

Image-objekter kan være forskelligt repræsenteret afhængig af type (GIF, JPG eller PNG) og opløsning

```
// fabrikeringsmetode leverer objekt til klienten
// this er et grafisk objekt, f.eks. applet, panel,..
Image i = this.getImage("billede.gif");  // korrekt
```

Designmønstret Singleton

(en klasse, der må være én og kun én instans af)

Problem: Klienten må ikke have flere objekter af en bestemt type, men skal altid bruge det samme objekt.

Løsning: Programmér sådan, at der aldrig kan oprettes mere end ét eksemplar af det pågældende objekt.

Eksempler

java.lang.Runtime (det kørende program)
java.awt.Toolkit (implementationen af grafiksystemet/AWT)

```
Runtime rt = Runtime.getRuntime();
// eksempler på brug af Runtime-objektet
System.out.println("Hukommelse reserveret til Java: "+rt.totalMemory());
System.out.println("Heraf ledigt: "+rt.freeMemory());
rt.gc(); // kør garbage collector
System.out.println("Nu ledigt: "+rt.freeMemory());
```

```
Toolkit tk = Toolkit.getDefaultToolkit();
// eksempler på brug af Runtime-objektet
System.out.println("Skærmstørrelse (punkter): " + tk.getScreenSize());
tk.beep(); // computeren siger bip
Image i = tk.getImage("billede.gif"); // her fungerer Toolkit som fabrik
```

Implementering af Singleton

- Normal implementering
  - Privat konstruktør
  - Instans oprettes ved klasseindlæsning og gemmes i privat klassevariabel
  - Fabrikeringsmetode returnerer den private instans

```
public class Dataforbindelse
{
   private static Dataforbindelse instans = new Dataforbindelse();

   public static Dataforbindelse hentForbindelse() { return instans; }

   private List alle;
   private Dataforbindelse() { alle = new ArrayList(); }

   public void sletAlleData() { alle.clear(); }
   public void indsæt(Kunde k) { alle.add(k); }
   public List hentAlle() { return alle; }
}
```

```
   Dataforbindelse1 dbf = Dataforbindelse.hentForbindelse();
   dbf.indsæt( new Kunde("Kurt",1000) );
```

Nedarvinger af Dataforbindelse?

Implementering af Singleton

- Normal implementering
  - Privat konstruktør
  - Instans oprettes ved klasseindlæsning og gemmes i privat klassevariabel
  - Fabrikerings(klasse)metode returnerer den private instans

- Andre implementeringer
  - Instans oprettes først når fabrikeringsmetode kaldes første gang
    - Fabrikeringsmetode må tjekke om instans allerede er oprettet
      - Trådsikkerhed kan blive et problem - fabrikeringsmetode skal være synchronized
  - Ingen fabrikeringsmetode, public final klassevariabel med instans
  - Ikke-privat konstruktør
    - Tillader nedarving
    - Konstruktør må tjekke om instans allerede er oprettet
      - Kast undtagelse hvis instans allerede findes
      - Trådsikkerhed kan blive et problem - konstruktør skal være synchronized
  - Fabrikeringsmetode i anden klasse
    - Konstruktør med pakke-synlighed

Andre implementeringer af Singleton

```
// Instans oprettes først når fabrikeringsmetode kaldes første gang
public class Dataforbindelse
{
   private static Dataforbindelse instans = null;

   public static synchronized Dataforbindelse hentForbindelse() {
     if (instans == null) instans = new Dataforbindelse();
     return instans;
   }
```

```
// Ingen fabrikeringsmetode, public final klassevariabel med instans
public class Dataforbindelse
{
   public static final Dataforbindelse instans = new Dataforbindelse()
```

```
// Ikke-privat konstruktør
public class Dataforbindelse
{
   public static Dataforbindelse instans = null;

   public static synchronized Dataforbindelse hentForbindelse() {
     if (instans == null) instans = new Dataforbindelse();
     return instans;
   }

   protected Dataforbindelse() {
     if (instans != null) throw new IllegalAccessException("Obj findes");
     instans = this;
```

Eksempel: Dataforbindelse

- Indkapsl datalagring i klasse
- Hvis man ønsker fleksibilitet omkring hvor data lagres
  - Start: DataforbindelseDummy, DataforbindelseFil
  - Slut: DataforbindelseOracle, ...

Serializable
Kunde
+navn:String
+kredit:double
+Kunde
+toString:String

Dataforbindelse
-forb:Dataforbindelse
+sletAlleData:void
+hentAlle:List
+hentForbindelse:Dataforbindelse

HttpServlet
DataforbindelseServlet
+init:void
+doGet:void

DataforbindelseOracle
-forb:Connection
-stmt:Statement
+DataforbindelseOracle
+sletAlleData:void
+hentAlle:List

DataforbindelseFil
-alle:List
-gem:void
+DataforbindelseFil
+sletAlleData:void
+hentAlle:List

DataforbindelseDummy
+sletAlleData:void
+hentAlle:List

DataforbindelseOverNetvaerket...
-basisUrl:String
+sletAlleData:void
+hentAlle:List

DataforbindelseCache
-df:Dataforbindelse
-cache:List
+DataforbindelseCache
+sletAlleData:void
+hentAlle:List

Designmønster Abstrakt Fabrik / Toolkit

(Fabrik med abstrakt superklasse og nedarvinger, som tager sig af oprettelsen)

Problem: En Fabrik bliver uforholdsmæssigt kompliceret, fordi nogle ydre omstændigheder har stor indflydelse på, hvordan oprettelsen skal foregå.

Løsning: Lav en Abstrakt Fabrik (eng.: Abstract Factory) med en nedarving (Fabrik) for hver omstændighed.

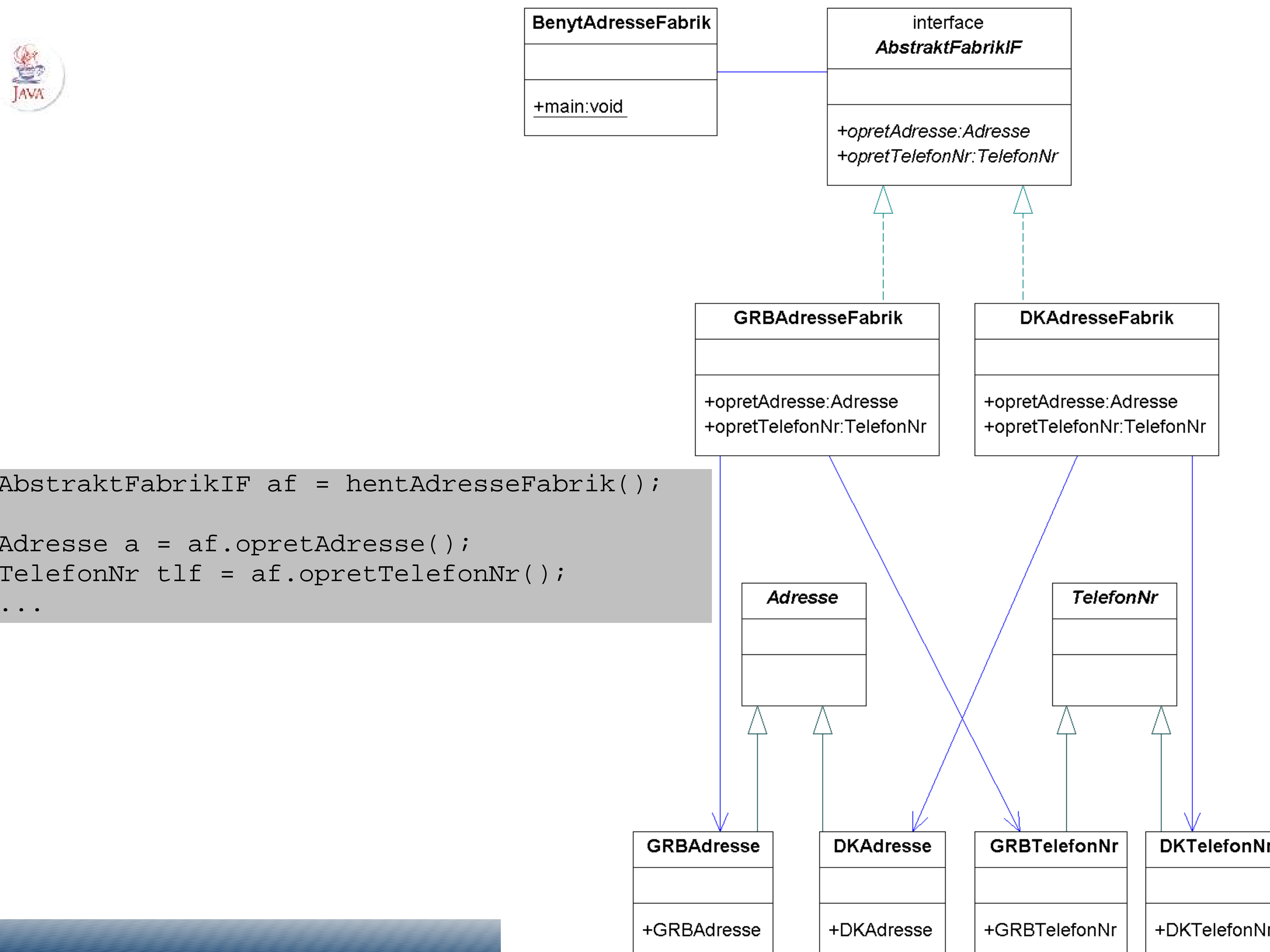
Eksempel: java.awt.Toolkit

Fabrikerer platformsspecifik del af AWT-komponent (peer)

Nedarvinger: WindowsToolkit, LinuxToolkit, SolarisToolkit

```
// følgende gøres i f.eks. java.awt.Button (aldrig fra normalt program!)
package java.awt;
public class Button extends Component {

   public Button() {
     Toolkit tk = Toolkit.getDefaultToolkit();
     ButtonPeer peer = tk.createButton(this); // platformsspecifik del!
```


BenytAdresseFabrik
+main:void
interface
AbstraktFabrikIF
+opretAdresse:Adresse
+opretTelefonNr:TelefonNr
GRBAdresseFabrik
+opretAdresse:Adresse
+opretTelefonNr:TelefonNr
DKAdresseFabrik
+opretAdresse:Adresse
+opretTelefonNr:TelefonNr
AbstraktFabrikIF af = hentAdresseFabrik();
Adresse a = af.opretAdresse();
TelefonNr tlf = af.opretTelefonNr();
...
Adresse
TelefonNr
GRBAdresse
+GRBAdresse
DKAdresse
+DKAdresse
GRBTelefonNr
+GRBTelefonNr
DKTelefonNr
+DKTelefonNr

Designmønstret Prototype

(objekter oprettes ud fra en skabelon)

Problem: Klienten ved ikke, hvad der skal oprettes, men kan dog angive et andet objekt, som ligner det, der skal oprettes

Løsning: Brug det andet objekt som Prototype, og opret objektet ud fra prototypen

Prototyper i et tegneprogram

- Palette har liste af figur-prototyper
  - Liste kan senere nemt udvides
- Bruger kan vælge i listen
  - Når der vælges i paletten, anvendes det pågældende element som Prototype til objektet, der skal tegnes på skærmen

```
Palette
-figurer:List=new ArrayList()
-navne:List=new ArrayList()
+Palette()
+tilfoej(navn:String,f:Figur):void
+navneliste():List
+opretFigur(indeks:int,x:int,y:int

0..*

Cloneable
Figur
x:int
y:int
+Figur(x1:int,y1:int)
+tegn(g:java.awt.Graphics):void
+kopi():Figur

JFrame
Figurtegning
palette:Palette=new Palette()
musPos:Point=new Point(0,0)
figurerPaaSkaermen:ArrayList=n
+paint(g:Graphics):void
this_mouseReleased(e:MouseEve
this_mouseDragged(e:MouseEve

Cirkel
radius:int
+Cirkel(x1:int,y1:int,radius1:int)
+tegn(g:java.awt.Graphics):void

Firkant
bredde:int
hoejde:int
+Firkant(x1:int,y1:int,bredde1:i
+tegn(g:java.awt.Graphics):void
```

Designmønstret Objektpulje

Problem: Der er et begrænset antal resurser, som skal deles.

Problem: Der oprettes for mange objekter. Programmet er langsomt eller kører ujævnt, fordi der oprettes så mange objekter, der løbende smides væk igen.

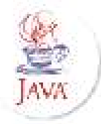
Objekterne kunne egentligt godt genbruges i stedet for at blive smidt væk, men oprettelsen sker spredt rundt i programmet, så det er svært at koordinere.

Løsning: Lad et objekt varetage resurserne/objekterne. Lad klienter reservere og frigive objekter gennem dette objekt.

```
public class Objektpulje
{
   private ArrayList ledige = new ArrayList();

   public synchronized void sætInd(Object obj) { ledige.add(obj); }

   public synchronized Object tagUd() {
     if (ledige.isEmpty()) throw new RuntimeException("Ikke flere objekter!");
     Object obj = ledige.remove(ledige.size()-1); // tag objekt ud af puljen
     return obj;
   }
}
```



Designmønstret Objektpulje

Andre muligheder hvis puljen løber tør for objekter

- Lad puljen oprette nye objekter (evt. v.hj.a. en Fabrik): Øvelse
- Lad klient 'hænge' og vente på at et objekt bliver ledigt:

```
public class ObjektpuljeKlientHaenger
{
   private ArrayList ledige = new ArrayList();

   public synchronized void sætInd(Object obj) {
     ledige.add(obj);
     this.notify(); // væk eventuelle ventende tråde
   }

   public synchronized Object tagUd() {
     try {
       while (ledige.isEmpty()) // så længe der ikke er ledige objekter...
       {
         System.out.println("Ikke flere objekter i puljen, venter...");
         this.wait();           // .... vent på at blive vækket
       }
       Object obj = ledige.remove(ledige.size()-1); // tag objekt ud af puljen
       return obj;
     } catch (InterruptedException e) { return null; }
   }
}
```

Gruppearbejde

- Hvilke af de designmønstrene ville/kunne I anvende i HAS-IT?
  - Fabrikeringsmetode
  - Singleton
  - Prototype
  - Objektpulje
- Diskutér to og to
  - Fordele og ulemper
  - Hvordan skulle klassediagrammerne ændres?
    - Mere eller mindre komplekst?
    - Mere eller mindre forståeligt?

Andre populære designmønstre

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;
   }
```

Jacob	-1899.0
Søren	600.0

Designmønstret Observatør/Lytter

(eng.: Observer/Listener)
eller Abonnent (eng.: Publisher-Subscriber)

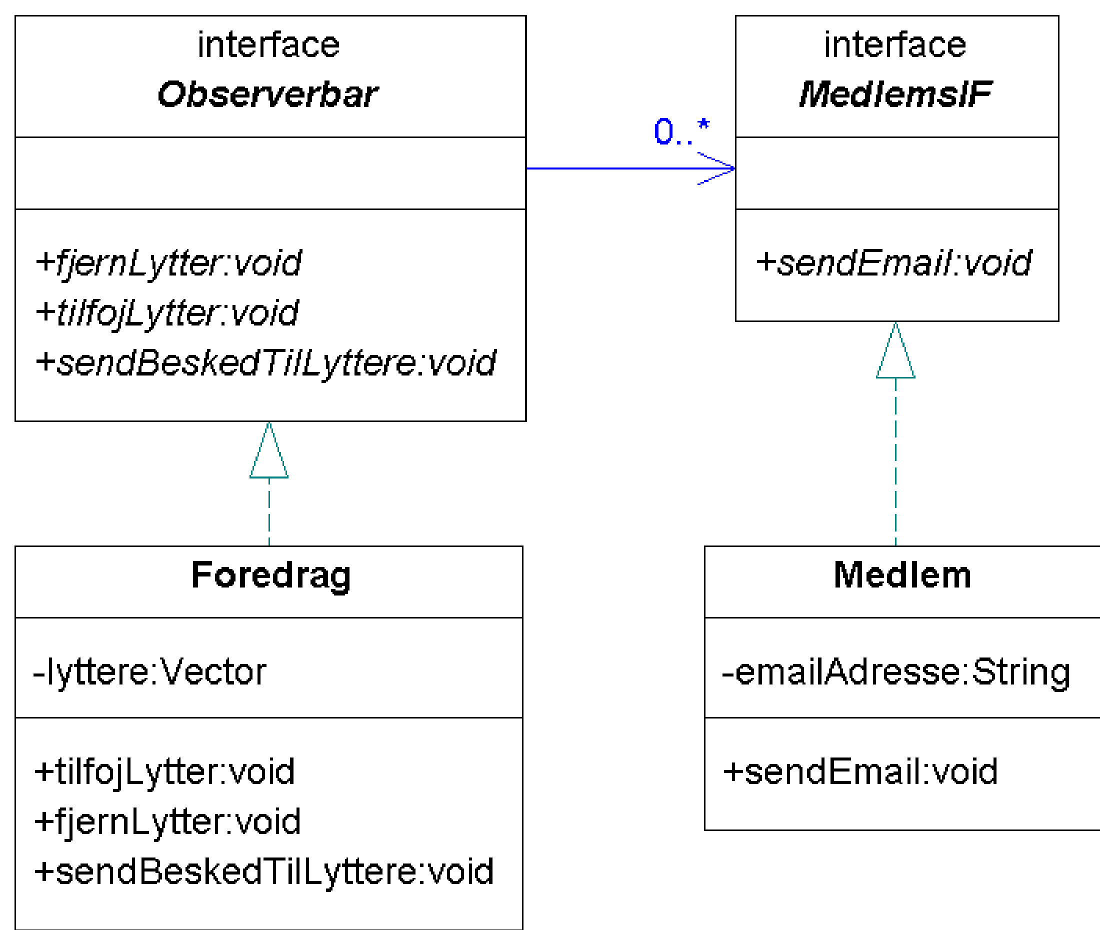
('abonnere' på, at en ting (hændelse) sker)

Problem: Et objekt skal kunne underrette nogle andre objekter om en eller anden ændring eller hændelse, men det er ikke hensigtsmæssigt, at objektet kender direkte til de andre objekter.

Løsning: Lad lytterne (observatørerne) implementere et fælles interface (eller arve fra en fælles superklasse) og registrere sig hos det observable (observerbare) objekt. Det observable objekt kan herefter underrette lytterne gennem interfacet, når der er brug for det.

Designmønstret Observatør/Lytter

- Eksempel: Fælles kalender for en forening, der udbyder foredrag
  - Hvis der sker ændringer i planen, skal kun de interesserede medlemmer have besked, dvs. de, som har tilmeldt sig et givent foredrag.
- xxx Interface Observerbar slettes, MedlemsIF omdøbes til Observatoer



Designmønstret Observatør/Lytter

- Programkode til observabelt objekt (og javabønne)

```
public class MinButton extends Component
{
  /** Lyttere til denne bønne */
  private ArrayList lyttere = new ArrayList(2);

  public void addActionListener(ActionListener l) { ly

  public void removeActionListener(ActionListener l) {

  /** Sender en hændelse til lyttere. Lytterne er de,
   *  tilføjet med kald til addActionListener() */
  protected void underretLytterne(ActionEvent hændelse
  {
    for (Iterator i=lyttere.iterator(); i.hasNext();
```

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;
   }
}
```

interface
Stak
+tagAf:Object

java.util.Vector
StakMedNedarving2
+tagAf:Object

Staklogger
-rigtigeStak:StakMedNedarving2
+Staklogger
+tagAf:Object

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

```java
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).
- Adgangsproxy - 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.*

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

Gruppearbejde

- Hvilke af de designmønstrene
  ville I/kunne I anvende/ER allerede anvendt
  i HAS-IT?
  - Proxy
  - Adapter
  - Observatør/lytter
  - ... (f.eks. Facade)
- Diskutér to og to
  - Fordele og ulemper
  - Hvordan skulle klassediagrammerne ændres?
    - Mere eller mindre komplekst?
    - Mere eller mindre forståeligt?

Opgave til næste gang

- Hvad overvejer du at lave i projektopgaven i OOM?
  - Brug 6 ½ time på det (det er den tid du forventes at lave hjemmearbejde i kurset ud over den 3 ½ times undervisning)
- Send mindst 5 linjers tekst (og højest 5 sider)
  - Slutbrugerens ønsker (=formålet med projektet)
    - Kort beskrivelse: Hvilket behov skal systemet opfylde? (i f.eks. HAS-IT er det 'at kunne få reguleret lamper, radiatorer, gardiner etc i sit hus automatisk')
  - Hvis tid: Udkast til overordnet kravspecifikation (kig f.eks. på opgaven i fildeling uge 2 i Campusnet)
- Beskrivelsen sendes søndag den 10/4 klokken 18 til: nordfalk@mobilixnet.dk
  - Til undervisningen får du ca. 5 minutter til præsentere din idé for de andre.
  - Overvej om du vil lave det i en gruppe med andre.