Videregående programmering i Java

Dag 9 - Andre designmønstre

Andre designmønstre: Uforanderlig, Fluevægt, Lagdelt Initialisering, Komposit/Rekursiv Komposition, Kommando/Ændring
Fremlæggelse af programmering/status for projekter
Evt.: Brug af Properties-filer (og præferencer i JDK 1.4)
Evt: Kommunikation med udklipsholder og Drag'n'drop

Læsning: VP 18, How to Use Drag and Drop and Data Transfer.

Hvordan går det med projektet?

- Præsentation af projekterne om 14 dage
  - Send (og medbring) klassediagram og skærmbilleder

Opgave til i dag

- Lav Trekant og FigurTekst i tegneprogramme
  - Lav 3-mandsgrupper
  - Præsentér jeres løsninger

Designmønstret Uforanderlig

Problem: Hvis der er flere referencer til et objekt, hvordan sikres det, at objektet ikke ændres gennem en af referencerne, sådan at de andre, der refererer til objektet, også utilsigtet får ændret 'deres' objekt?

Løsning: Lav objektet uforanderligt (eng.: Immutable) ved at sørge for, at det er umuligt at ændre i objektets data. Alle forsøg på at 'ændre' objektet resulterer i nye objekter.

- Et uforanderligt objekt er et objekt, der ikke kan ikke ændres, når det først er oprettet.
  - Vigtigste eksempel i standardbiblioteket: String
- Eventuelle ændringer sker ved at oprette nye objekter
  - Giver lidt speciel 'syntaks': **s** = s.toUpperCase()

Designmønstret Uforanderlig

- Eksempel:

```
public class UforanderligtPunkt
{
   private double x,y;

   public UforanderligtPunkt(double x1, double y1)
   {
      x = x1;
      y = y1;
   }

   public UforanderligtPunkt flytRelativ(double dx, double dy)
   {
      UforanderligtPunkt nytPkt = new UforanderligtPunkt(x+dx, y+dy);
      return nytPkt;
   }

   public double getX()
   {
      return x;
   }

   public double getY()
   {
      return y;
   }

   public double afstand(UforanderligtPunkt p)
   {
      return Math.sqrt( (x-p.x)*(x-p.x) + (y-p.y)*(y-p.y) );
   }
}
```

- Opgave:
  Implementér
  setX(double nyX)
  setY(double nyY)

Designmønstret Fluevægt

Problem: Hvordan reduceres hukommelsesforbruget, når der haves mange objekter med den samme eller næsten den samme information?

Løsning: Lav flere referencer til de samme objekter, og hold styr på, hvilke objekter der allerede er oprettet. Lad omgivelserne holde styr på eventuelle ændringer.

- Begræns antallet af objekter ved at sørge for, at der ikke bliver oprettet objekter med de samme data ved at dele dem, sådan at der i stedet bliver mange referencer til de samme (unikke) objekter
- En klasse, der deles på denne måde, siges at være en Fluevægt (eng.: Flyweight).
  - Vigtigste eksempel i standardbiblioteket: s = s.intern();

Designmønstret Fluevægt

- Eksempel: UforanderligtPunktFabrik

```
import java.util.*;

/** Fabrik til UforanderligtPunkt.
    Behandler UforanderligtPunkt-objekterne som Fluevægte, dvs. det samme
    objekt kan blive delt hvis det indeholder de samme data.
    Implementationen er ret ineffektiv, da fabrikationen er O(antal punkter)
 */
public class UforanderligtPunktFabrik
{
    private List punkter = new ArrayList();

    public UforanderligtPunkt opretPunkt(double x, double y) {
      UforanderligtPunkt p;

      // se om der allerede findes et punkt med disse koordinater
      for (Iterator i=punkter.iterator(); i.hasNext(); ) {
        p = (UforanderligtPunkt) i.next();
        if (p.getX()==x && p.getY()==y) return p;
      }

      // ingen fundet - opret nyt
      p = new UforanderligtPunkt(x,y);
      punkter.add(p);
      return p;
    }
}
```

Designmønstret Lagdelt Initialisering

Problem: Klienten ved ikke præcist, hvad der skal oprettes og hvordan, men skal alligevel kunne initialisere (oprette) objekter direkte (med new Klassenavn())

Løsning: Lad initialiseringen være lagdelt: Klienten initialiserer et objekt direkte (med new Klassenavn()), og dette objekt opretter eller fremskaffer det, der i virkeligheden skal bruges, og delegerer efterfølgende kald videre.

Lagdelt Initialisering (eng.: Layered Initialization) er endnu en løsning på problemet med at vælge en specialiseret klasse til en opgave, uden at 'klienten' - brugeren af klassen - behøver at tage stilling til, præcis hvilken specialiseret klasse der skal oprettes.

Designmønstret Lagdelt Initialisering

- Lad klienter oprette objekter fra en 'tynd' klasse, der selv beslutter hvilke hjælpeklasser den har brug for, opretter dem og delegerer det meste af arbejdet ud til dem
- I Lagdelt Initialisering vil objektet, der oprettes af klienten, ofte selv tager sig af den generelle logik, mens den specialiserede logik vil være delegeret ud til de specialiserede klasser.
- Med Lagdelt Initialisering:

```
Hjælp      h = new Hjælp();
List       l = new ArrayList();
URL        u = new URL("http://javabog.dk");
DateFormat f = new SimpleDateFormat("dd/MM yyyy 'klokken' hh:mm");
```

- Uden Lagdelt Initialisering:

```
Hjælp      h = Hjælp.opretHjælp();
List       l = Collections.createList();                    // fiktivt eksempel
URL        u = URL.createURL("http://javabog.dk");          // fiktivt eksempel
DateFormat f = DateFormat.getTimeInstance(DateFormat.FULL);
```

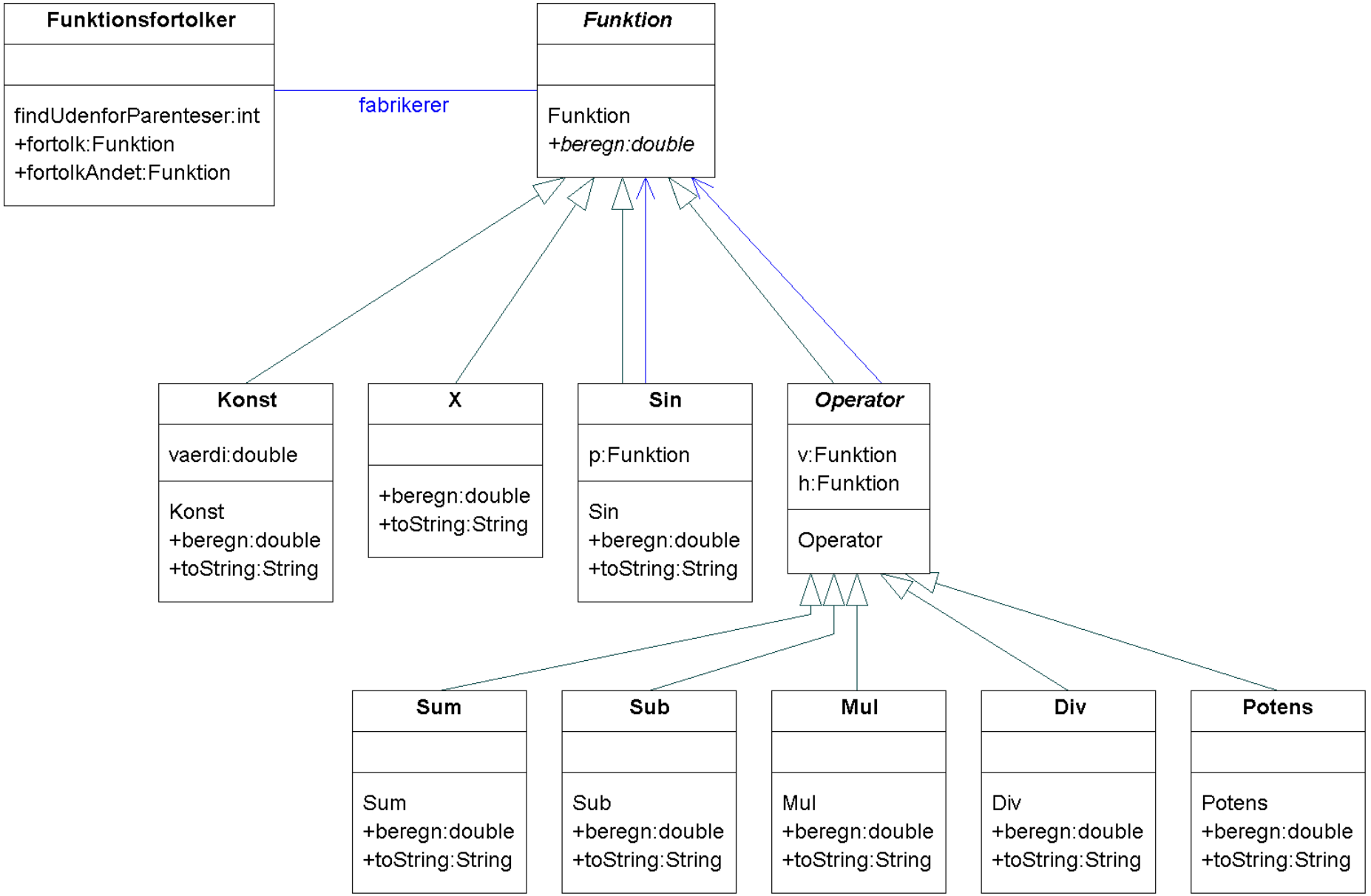
Designmønstret Komposit

(eng.: Composite; objekt kan indeholde andre objekter, inkl. sin egen slags)

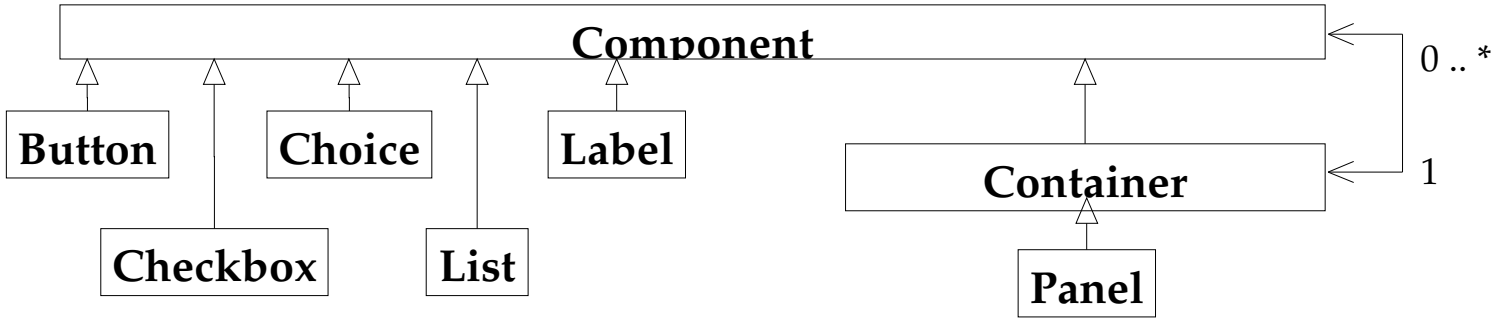
Problem: Hvordan kan man gruppere nogle objekter, så de kan behandles som ét objekt? Der er brug for grupperinger i flere niveauer

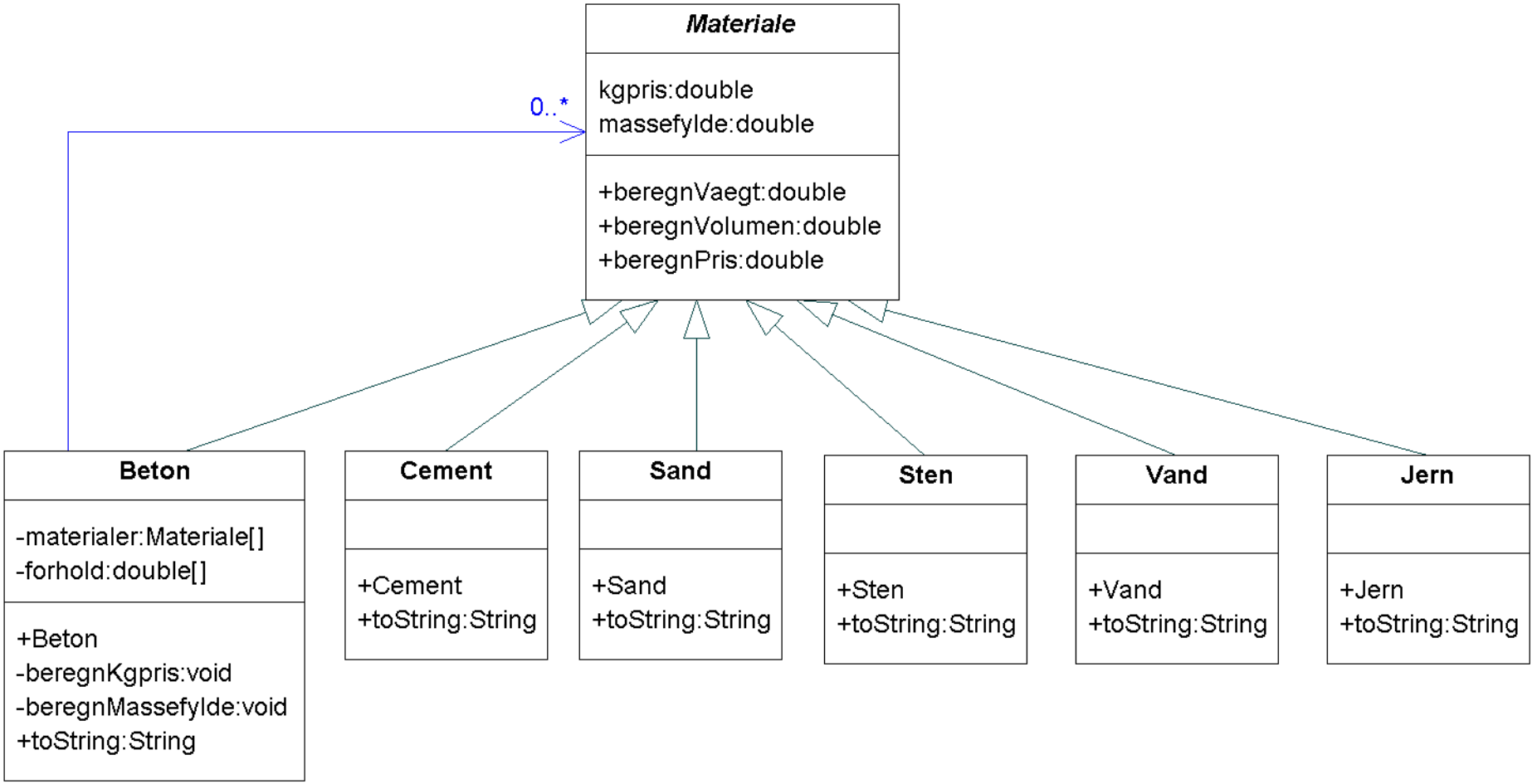
Løsning: Definér en fælles superklasse for alle enkeltobjekter, og definér en klasse til sammensatte objekter, som har en liste af de objekter den består af

Designmønstret Komposit/Rekursiv Komposition



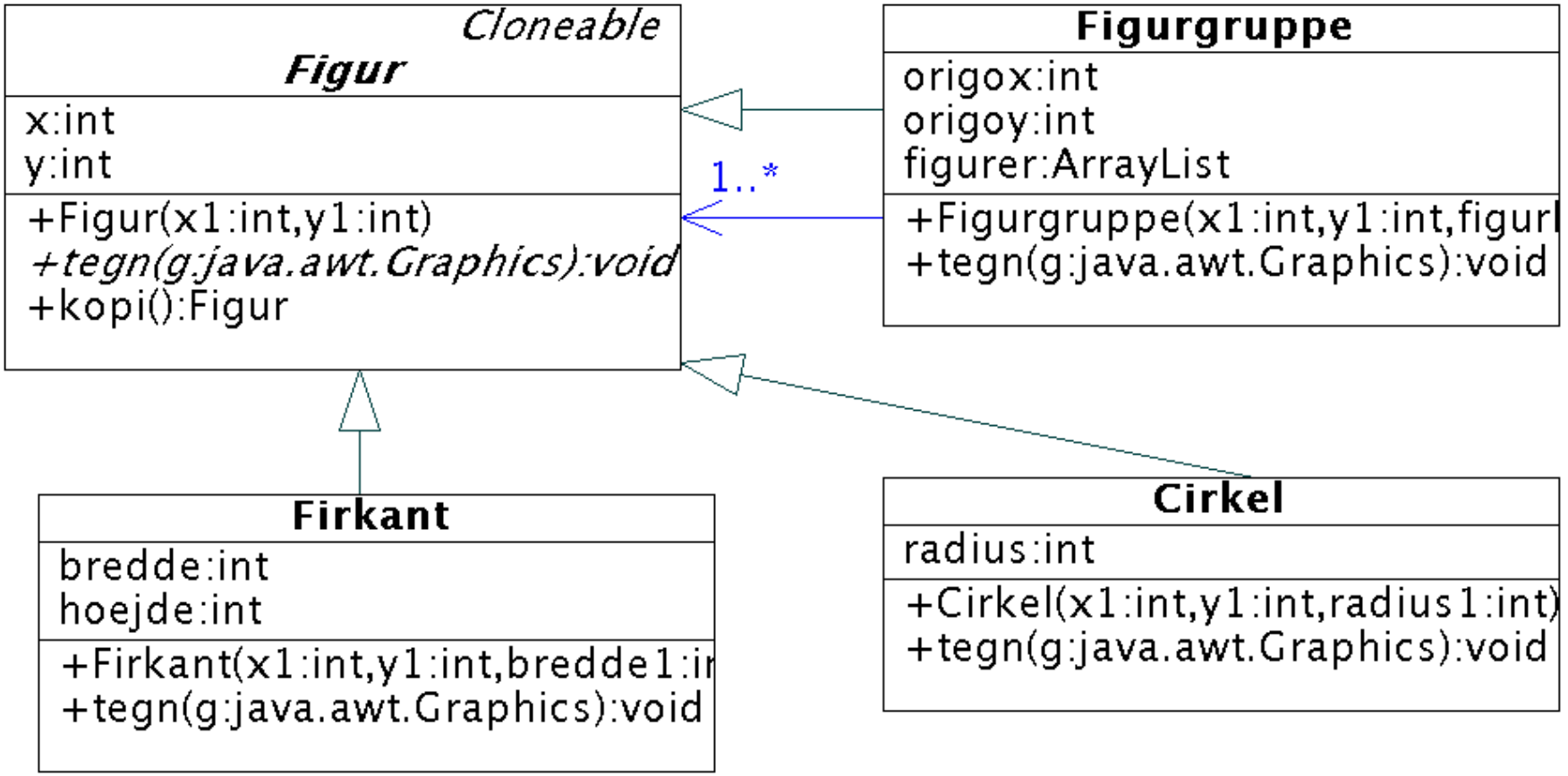
Designmønstret Komposit/Rekursiv Komposition

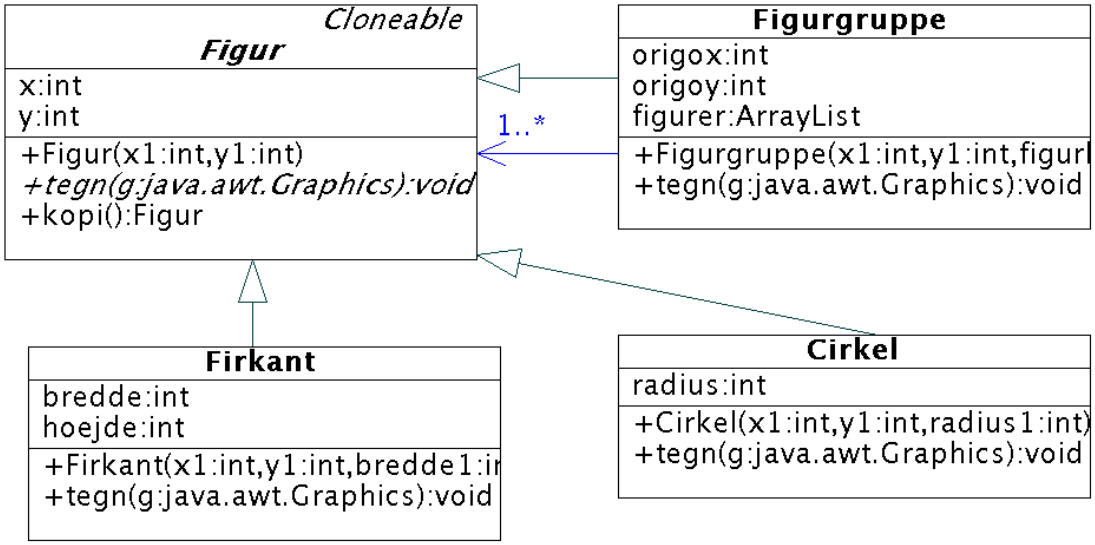




Komposit i et tegneprogram

- Figurgruppe
  - Arver fra Figur
  - Har en liste af figurer






```
import java.util.*;

public class Figurgruppe extends Figur
{
    int origox;
    int origoy;

    ArrayList<Figur> figurer;

    public Figurgruppe(ArrayList<Figur> figurliste)
    {
       super(0,0);
       figurer = new ArrayList<Figur>(figurliste); // kopiér listen
       for (int i=0; i<figurer.size(); i++) {
          Figur f = (Figur) figurer.get(i);
          origox = origox + f.x;
          origoy = origoy + f.y;
       }
       origox = origox / figurer.size();
       origoy = origoy / figurer.size();
    }

    public void tegn(java.awt.Graphics g)
    {
       g.translate(x-origox,y-origoy);
       for (int i=0; i<figurer.size(); i++) {
          Figur f = (Figur) figurer.get(i);
          f.tegn(g);
       }
       g.translate(origox-x,origoy-y);
    }
}
```

Designmønstret Kommando

(eng.: Command; registrér brugerens handlinger, så de kan fortrydes igen)

Problem: Brugeren skal kunne udføre og fortryde sine ændringer på data
(naiv løsning: Kopiér alle data ved hver ændring)

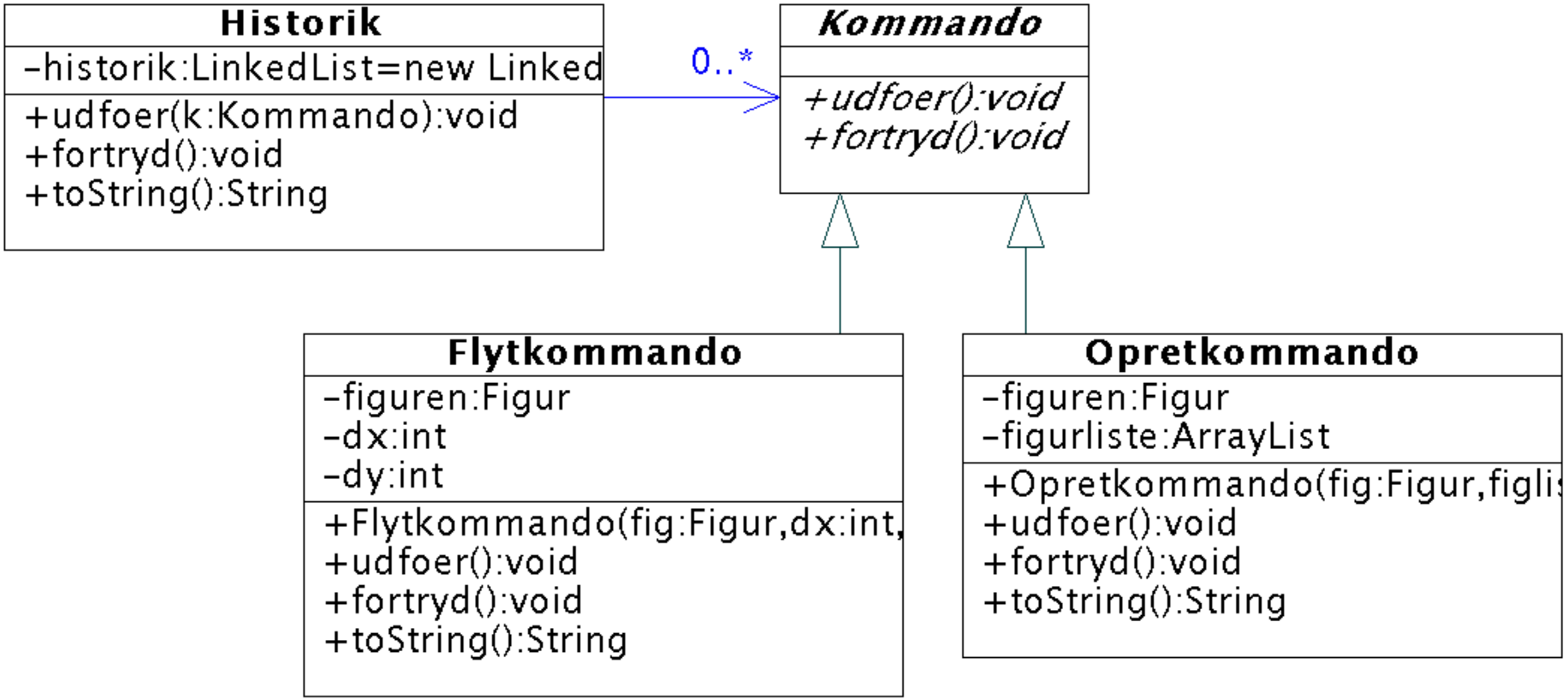
Løsning: Definér et Kommando-objekt, der repræsenterer ændringen

- ved, hvordan ændringen udføres
- husker de oprindelige værdier, sådan at ændringen kan fortrydes igen

- I stedet for Kommando (eng.: Command) kunne man også tale om Ændring-objekter eller Rettelse-objekter.

Kommando i tegneprogram

- Kommando med udfør og fortryd
  - FlytKommando(husker ændring i x og y)
  - OpretKommando
- Kommandoer gemmes i historik



Designmønstret Kommando/Ændring

- Registrér alle ændringer i data, sådan at de kan fortrydes igen, ved at definere Kommando-objekter for hver ændring
- Kommando-objektet har som minimum referencer til de objekter den skal ændre på og metoden udfør(), som udfører ændringerne.
- Ønsker man en fortryd-funktion, hvor brugeren kan fortryde (og evt. gendanne) de sidste handlinger, skal Kommando-objektet også have metoden fortryd() (og evt. også gendan()) og huske, hvordan objekterne så ud i den gamle tilstand.

Opgave

- Implementér Gendan i figurtegning-eksemplet
-

Evt.: Brug af Properties-filer

- God til at gemme (tekst)data, f.eks. vinduesplaceringer og brugernavne

```
String bn = "jano";
int x = 500;
Properties p = new Properties();     // husk: import java.util.*;
p.setProperty("brugernavn", bn);
p.setProperty("vindue.x", ""+x);
p.store(new FileOutputStream("figurtegning.properties"),"data for figurtegning");
```

- Data gemmes i navngivet fil

figurtegning.properties

```
#data for figurtegning
#Mon Nov 01 15:45:36 CET 2004
vindue.x=500
brugernavn=jano
```

  - Muligt at gemme sammen med klasserne (f.eks. i JAR-fil)
    - Class.getResourceAsStream ("figurtegning.properties");

- Brugeren behøver ikke taste data ind igen næste gang han starter programmet

```
Properties p = new Properties();
p.load(new FileInputStream("figurtegning.properties"));
bn = p.getProperty("brugernavn","gæst");
x  = Integer.parseInt(p.getProperty("vindue.x","0"));
```

Evt.: Brug af præferencer i JDK 1.4

- God til at gemme f.eks. vinduesplaceringer og brugernavne

```
String bn = "jano";
int x = 500;
Preferences.userRoot().node("figurtegning").put("brugernavn", bn);
Preferences.userRoot().node("figurtegning").putInt("vindue.x", x);
//evt.: Preferences.userRoot().sync();
```

- Data gemmes som XML under programmets navn

~/.java/.userPrefs/figurtegning/prefs.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE map SYSTEM 'http://java.sun.com/dtd/preferences.dtd'>

<map MAP_XML_VERSION="1.0">
  <entry key="brugernavn" value="jano" />
  <entry key="vindue.x" value="500" />
</map>
```

- Brugeren behøver ikke taste data ind igen næste gang han starter programmet

```
bn = Preferences.userRoot().node("figurtegning").get("brugernavn","gæst");
x  = Preferences.userRoot().node("figurtegning").getInt("vindue.x",0);
```

Evt: Kommunikation med udklipsholder og Drag'n'drop