DD1385

Programutvecklingsteknik

Några bilder till föreläsning 6

Innehåll

- Mål för programutveckling
- Designkriterier
- Designmönster:
  - Template Method
  - Composite
  - Factory-teknik
  - Strategy
  - Relation
  - Proxy

Målen för programutveckling

Programprodukten bör vara

Korrekt - gör det den ska

Robust - tål att man ger fel indata

Flexibel - går att ändra med rimlig ansträngning

Återanvändbar - för att spara arbete (dvs pengar)

Effektiv - m.a.p. minne och processorkraft

Pålitlig - det tar lång tid innan programmet kraschar

Användbar - t.ex. begripligt GUI för användare

Designkriterier

Vägledning – **inte** lagar/stenhårda regler

- Inkapsling — Encapsulation
- Lös koppling — (Loose) Coupling
- Sammanhållning — Cohesion
- Ansvarsdrivet
- Återanvändbarhet på flera sätt
  - Färdiga komponenter
  - Klasser att ärva ifrån
  - Ramverk (t.ex. awt, swing, Collection)
  - Beprövad design: designmönster
- Gör Refactoring

Inkapslingsexempel: klassen `Point`

Klassen `Point` representerar en 2D-punkt i både kartesiska och polära koordinater.

`x, y, r, fi`

Klassen `Point` tillämpar inte inkapsling

Inkapslingsexempel, forts.

```
class Point {
    double x,y,r,fi;

    Point (double r, double fi) {
        this.r = r; this.fi = fi;
        x = r*Math.cos(fi);
        y = r*Math.sin(fi);
    }
}
```

Objekt av `Point` kan lätt göras **inkonsistenta** !

T.ex. genom `x` eller `y` ändras utan att `r` och `fi` uppdateras.

Kapsla in !

```
class Point1 { /*** SAFER VERSION ***/

    private double x,y,r,fi;

    Point1(...){...} // Konstruktor

    public void setPolar(double r, double fi) {
      // Update all x,y,r,fi correctly
    }

    public void setCart(double x, double y) {
      // Update all x,y,r,fi correctly
    }

    // get-methods for x,y,r,fi
}
```

Fullständiga Point1, början

```
class Point1 {

    private double x,y,r,fi;

    Point1 (double r, double fi){
        setPolar(r,fi);
    }

    public void setPolar(double r, double fi){
        this.r = r; this.fi = fi;
        x = r*Math.cos(fi);
        y = r*Math.sin(fi);
    }

    // to be continued ...
```

Fullständiga Point1, slutet

```
    public void setCart(double x, double y){
        this.x = x; this.y=y;
        r = Math.sqrt(x*x + y*y);
        fi = Math.atan2(y,x);
    }

    public double getX() {return x;}
    public double getY() {return y;}
    public double getR() {return r;}
    public double getFi() {return fi;}
}
```

Abstrakt datatyp (ADT)

- Data och deras lagringsformat är dolda
- Operationer är tillgängliga genom interface
  - Man får veta vad
  - Man får inte veta hur

Fördelar

- Säkerhet, data är skyddade
- Användaren behöver inte förstå ADT:ns insida
- Insidan av ADT:n kan förbättras/bytas utan att att användarprogram behöver ändras

Sammanhållning

En klass tar ansvar för allt som rör en sak
men inget annat.

Ansvarsdriven design

En klass tar ansvar för alla operationer
på sina egna data.

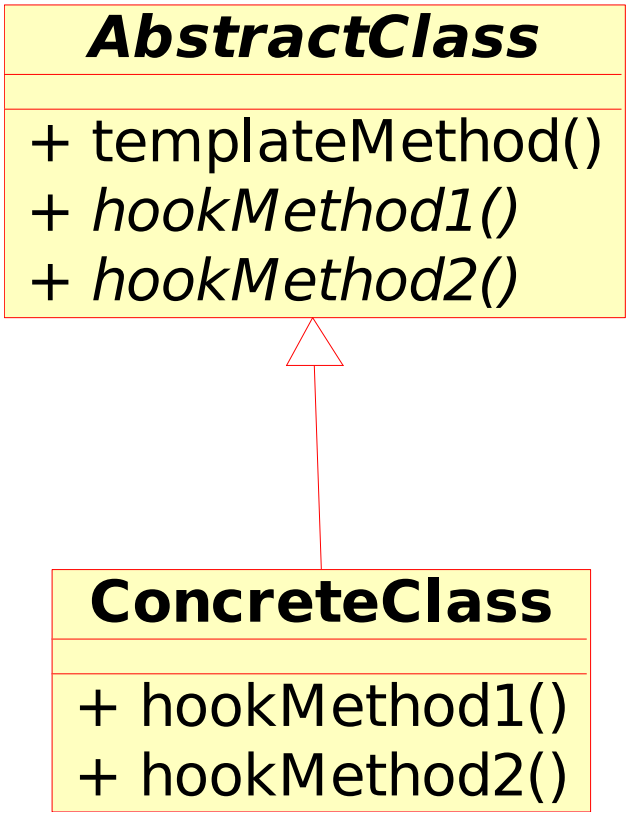
Lös koppling

Gäller mellan klasser: små och grunda gränssnitt,
gärna baserat på `interface`.

Designmönster: Template Method

Skjut upp delar av en algoritm till subklasser

Template Method

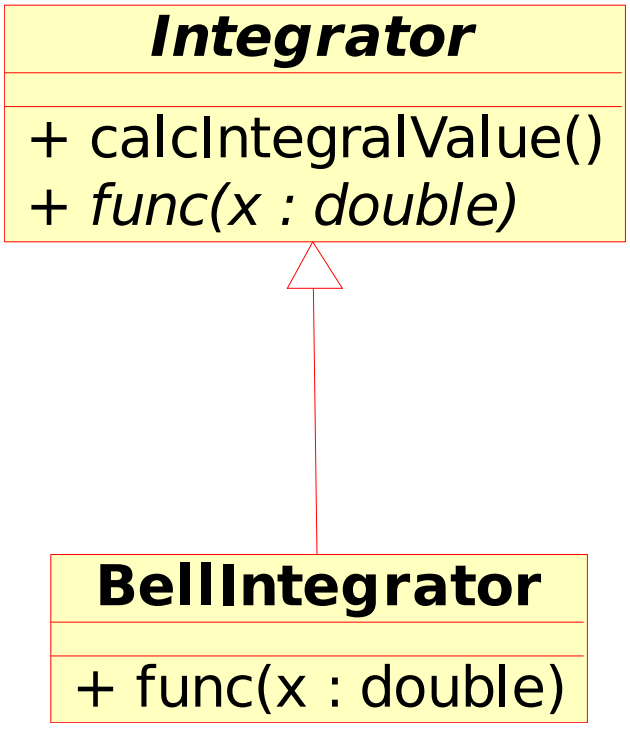


`templateMethod()`
beskriver en konkret algoritm

hookMethod1()
hookMethod2()
används i `templateMethod()`
men är abstrakta!

I `ConcreteClass`
definieras konkreta
`hookMethod1()`
`hookMethod2()`

Template Method – exempel



Integralen beräknas,
`func(x)` anropas.

men `func(x)` definieras
först här.

Template Method - exempel : Numerisk integrering

```
abstract class Integrator {
    :

  void set (double l, double h, double p){...}

  double calcIntegralValue() {
        :
    for (int i=0; i<n; i++){
          :
       sum += func(x);
    }
  }
  abstract double func(double x);
}
```

Vilken funktion integreras ?

Funktion att integrera bestäms i subklass

```
class BellIntegrator extends Integrator{
  double func(double x) {
    return Math.exp(-x*x);
  }
```

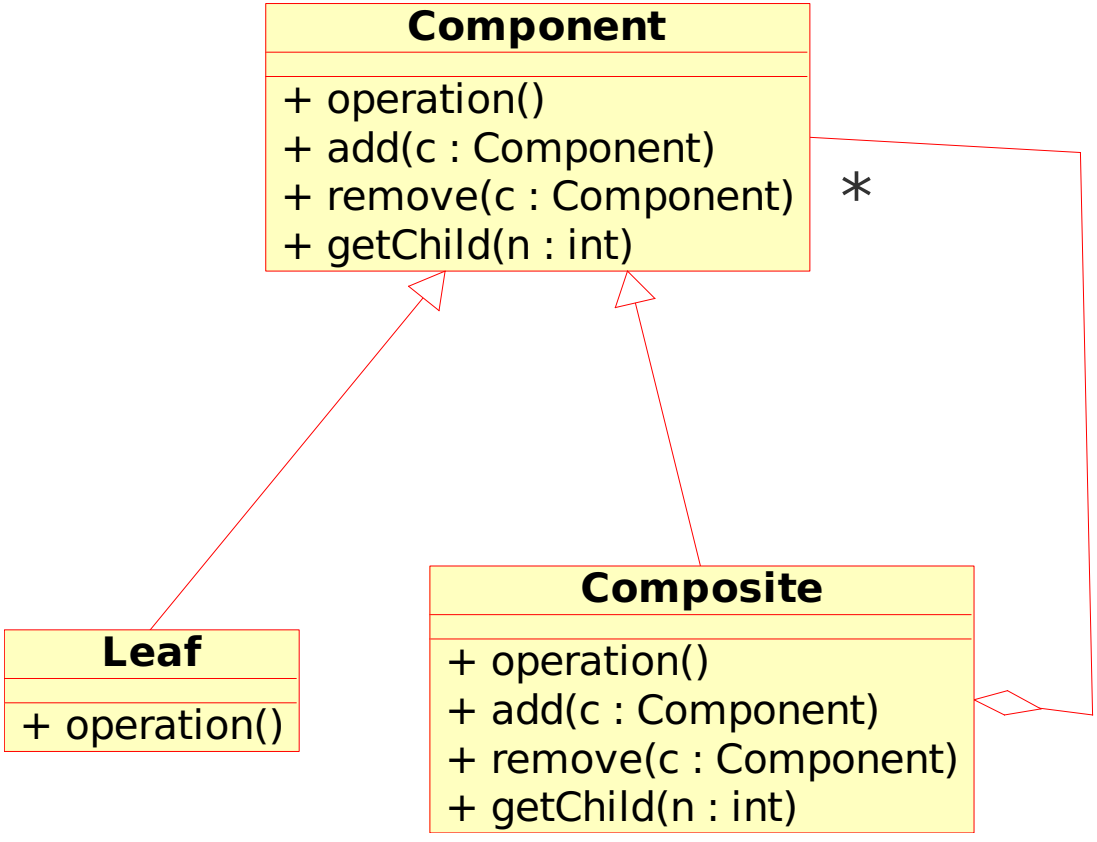
Testa i main-metod:

```
  public static void main(String[] a) {
    BellIntegrator bi = new BellIntegrator();
    bi.set(0.1, 0.5, 0.001);
    System.out.println(bi.integralValue());
    bi.set(0, 1, 0.001);
    System.out.println(bi.integralValue());
  }
}
```

Designmönster: Composite

- Objekt ordnas i en trädstruktur
- Operationer kan utföras på enskilda objekt eller grupper av objekt

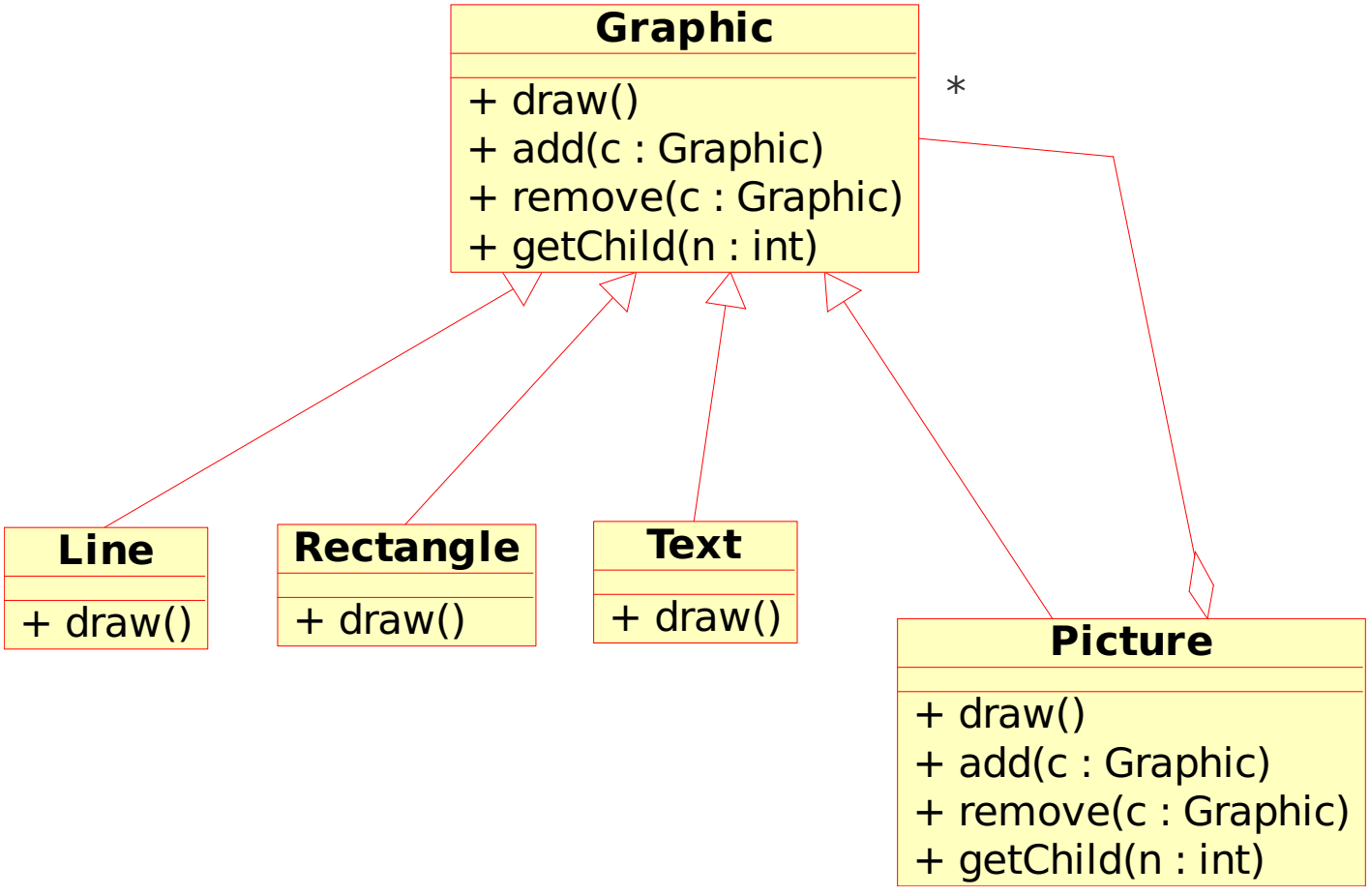
Composite



Composite
har en lista med Components

```
operation():
for (o:lista)
o.operation()
```

Composite-exempel



Factory-teknik

- Objekt skapas *inte* med
  `new A(...)`
- Objekt skapas med fabriksmetod, t.ex.
  `A.getInstance(...)`

Varför ?

Skäl för factory-teknik

- Säkerhet: - antalet objekt ska kontrolleras
- Säkerhet o/e bekvämlighet:
  - en särskild subklass ska användas
- Effektivitet
  - kanske inte nödvändigt att skapa nytt objekt
- Flexibilitet - t.ex. om flera konstruktorer med samma signatur önskas

```
new Point(x,y)
new Point(r,fi)
```

båda är Point(double, double)

```
class Point {
    private double x, y, r, fi;

    private Point (){}

    static Point createPolar
                (double r, double fi)
    {...}

    static Point createCartesian
                (double x, double y)
    {...}
    ......... // fler metoder
}
```

Fabriksmetoden createPolar

```
static Point createPolar(double r, double fi){
   Point p = new Point();
   p.r = r; p.fi = fi;
   p.x = r*Math.cos(fi);
   p.y = r*Math.sin(fi);
   return p;
}
```

Fabriksmetoden createCartesian

```
static Point createCartesian
                              (double x, double y){
  Point p = new Point();
  p.x = x; p.y = y;
  p.r = Math.sqrt(x*x + y*y);
  p.fi = Math.atan2(y,x);
  return p;
}
```

eller

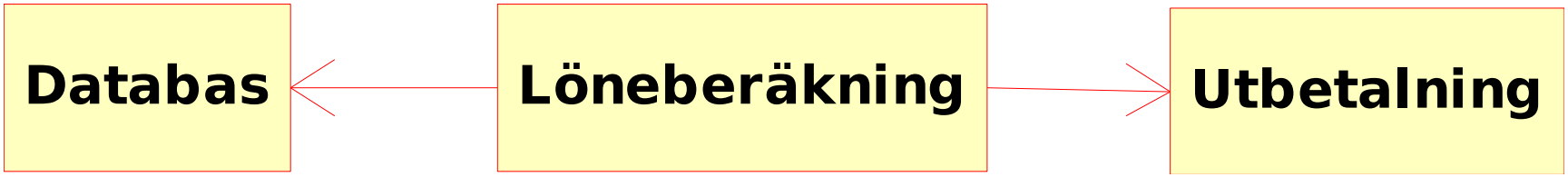
```
static Point createCartesian
                              (double x, double y){
  Point p = new Point();
  p.setCartesian(x,y);
  return p;
}
```

Skapa nya Point-objekt

```
Point p1 = Point.createCartesian (4.3 ,5.8);
Point p2 = Point.createPolar (6.0 ,0.6);
```

Exempel på lös koppling: mönstret Mock Object

Mock Object



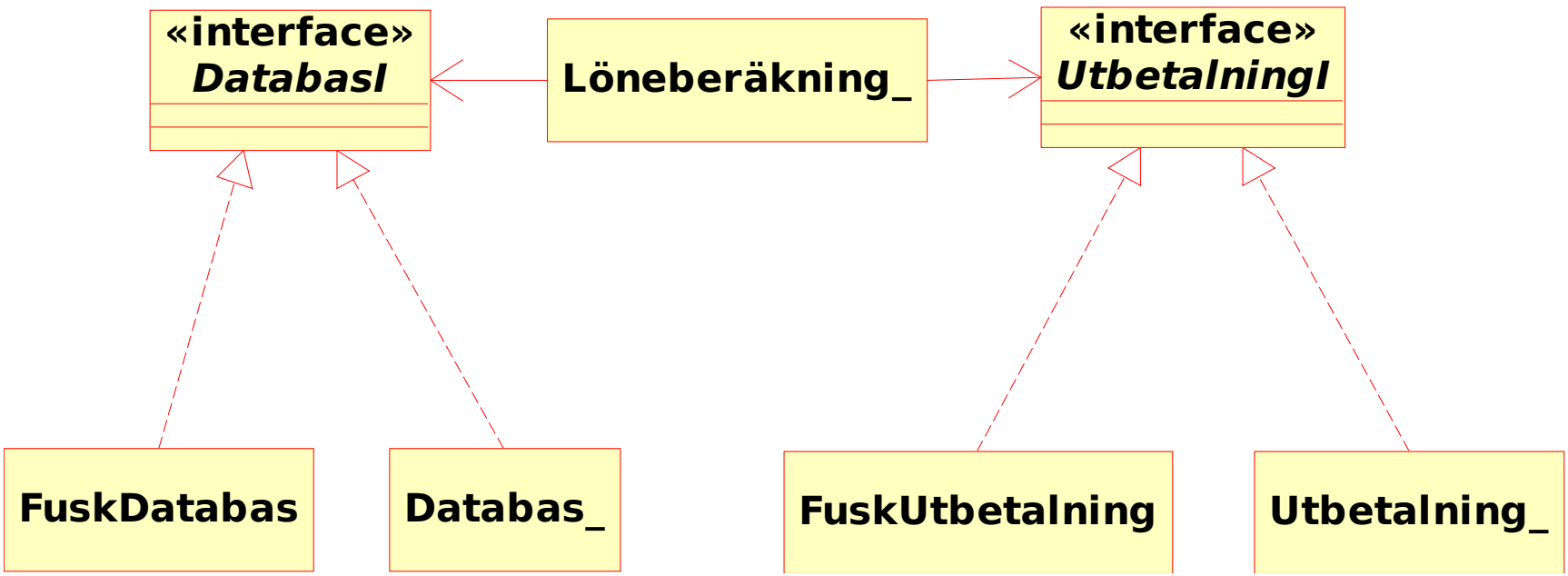
`Löneberäkning` beror "hårt" av

`Databas` och `Utbetalning`.

Både `Databas` och `Utbetalning` måste finnas innan man kan prova `Löneberäkning` !!

Mock Object – Lös koppling:

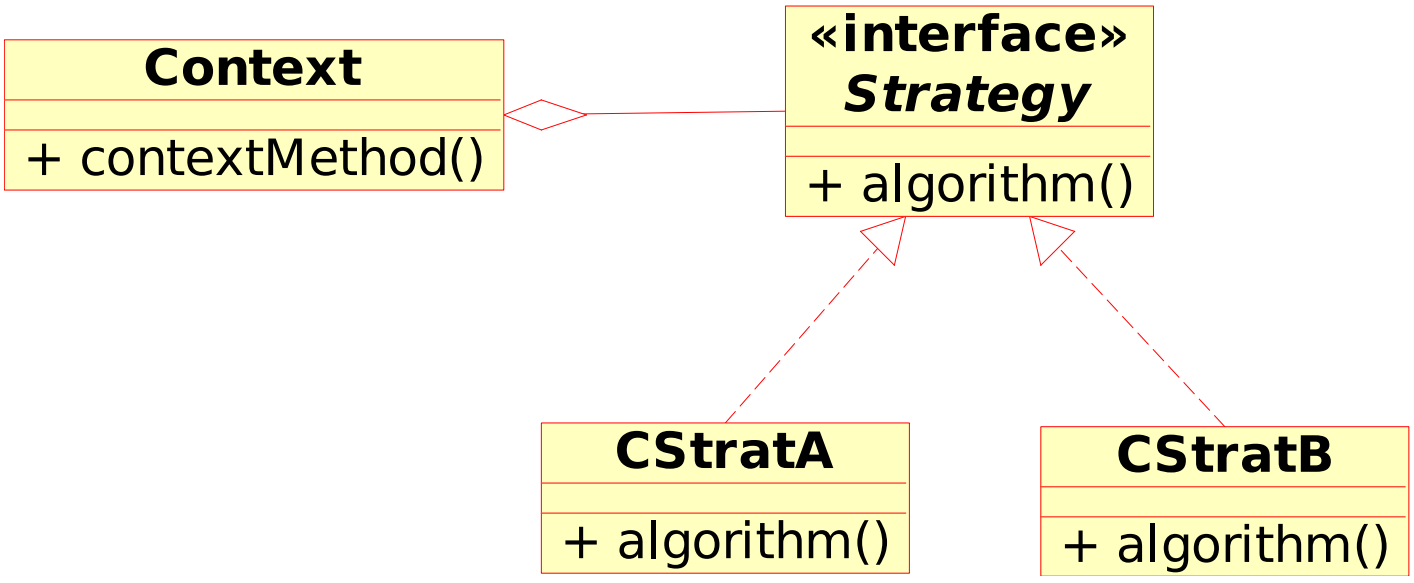
Låt `Löneberäkning` bero av interface istället för konkreta klasser.



Designmönster: Strategy

- En del av en algoritm är utbytbar
- Den utbytbara delen finns i ett objekt
- Den utbytbara delen definieras i ett interface
- Interfacet ges flera implementationer

Strategy



- Context-objektet har ett Strategy-objekt
- Metoden `contextMethod()` anropar `strategy.algorithm()`

Sortering av lista med Collections.sort

```
Collections.sort(list)
```

- Sorterar `list`
- Listans objekt `implements Comparable`
  dvs har metoden `compareTo(...)`
- Vid sorteringen jämförs elem1 och elem2:
  `elem1.compareTo(elem2)`
  ger -1, 0 eller 1

Strategy-exempel med Collections

- Sortera en lista
- Objekten behöver inte vara `Comparable`
  går det? JA
- `Collections.sort(list, coll)`

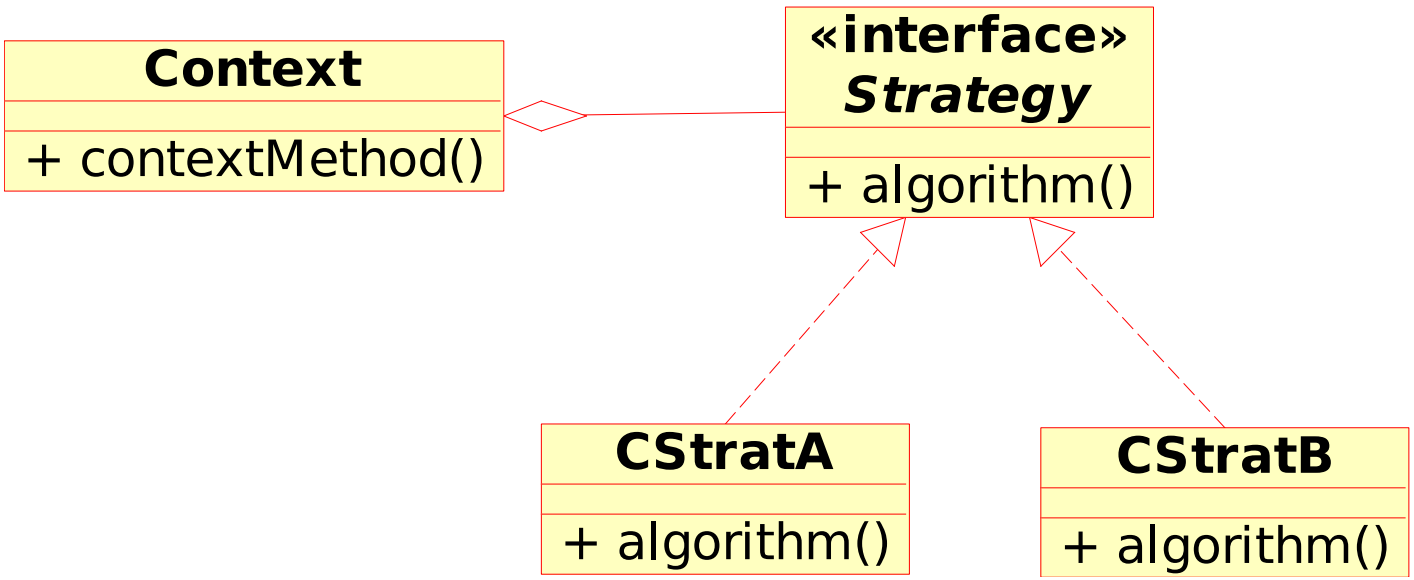
`Collections.sort(list, coll)`

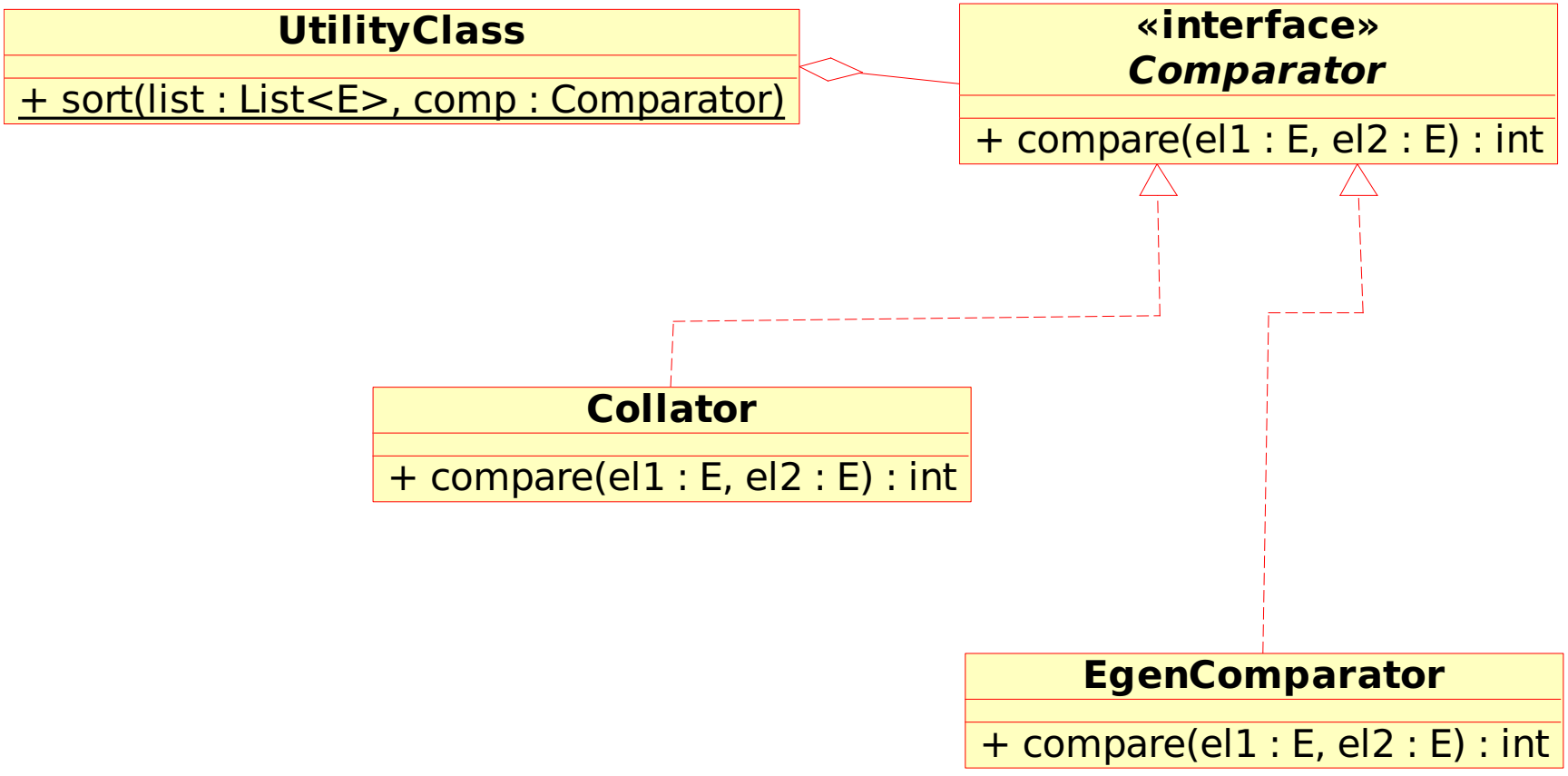
- Jämförelsefunktion kommer i objektet `coll`
- `coll` har typen `Comparator`
- ```
  interface Comparator<E> {
    int compare(E elem1, E elem2);
  }
  ```
- `compare()` returnerar något av `-1 0 1`

  beroende på ordningen mellan `elem1` och `elem2`
  ```

- `Comparator` - objektet är en utbytbar del av sorteringsalgoritmen
- Använd olika `Comparator` - implementationer beroende på sammanhang
- Mönstret Strategy !!!

Strategy


Context
+ contextMethod()
«interface»
Strategy
+ algorithm()
CStratA
+ algorithm()
CStratB
+ algorithm()



Collator är en bibliotekskklass för språkberoende textjämförelser.

Comparator-exempel

På övning 3 hade vi

```
class Person implements Comparable<Person>{
    long pnr; String namn;
    ....
}
```

jämförelsen görs på personnumret `pnr`

`Collections.sort(listOfPersons)`

sorterar i personnummerordning

Sortera på namn istället:

Egen `Comparator` för namnjämförelse:

```
class NamnComp implements Comparator<Person>{

   int compare(Person p1,Person p2){
      String n1 = p1.namn;
      String n2 = p2.namn;
      return n1.compareTo(n2);
   }
}
```

`String` är `Comparable` och har metoden `compareTo()`

`lista` innehåller Personobjekt.

Sortera i namnordning:

```
Collections.sort(lista, new NamnComp());
```

Tyvärr kommer å och ä i fel ordning

Ordnas med ett objekt av

Collator

Sortera svenska bokstäver rätt:

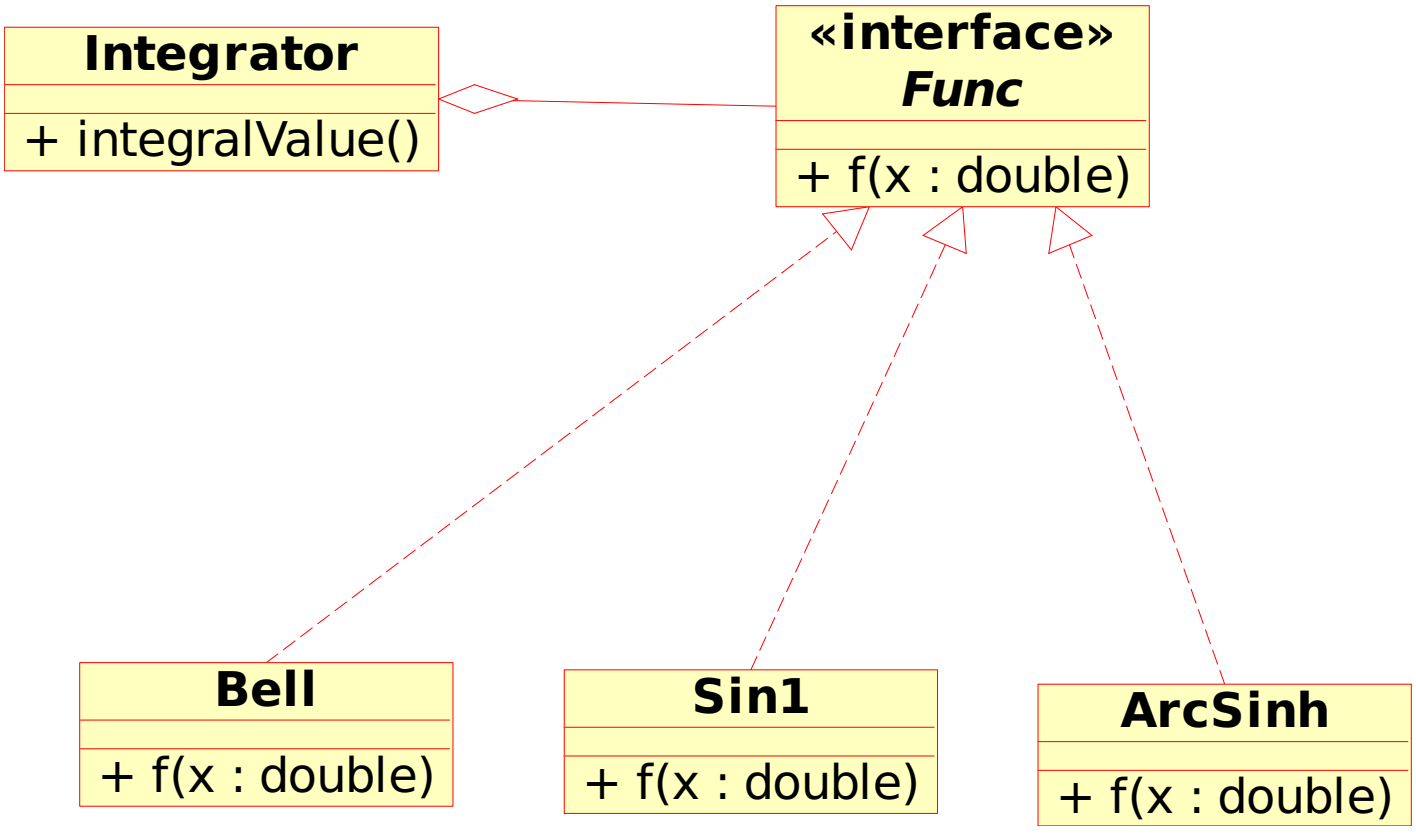
Egen `Comparator` för namnjämförelse:

```
class NamnComp implements Comparator<Person>{
  Collator collator = Collator.getInstance();
  int compare(Person p1,Person p2){
    String n1 = p1.namn;
    String n2 = p2.namn;
    return collator.compare(n1,n2);
  }
}
```

`Collator.getInstance()`
ger ett jämförelseobjekt för svenska (default).

Med parametrar ges objekt för andra språk, t.ex.
`Collator.getInstance(Locale.FRENCH)`

Strategy: funktion till Integrator-exemplet



Interface Func

Integratorn använder ett Func-objekt

dvs ett objekt av en klass som

implementerar Func

```
public interface Func {
    public double func(double x);
}
```

Skiss av Integrator-klassen, endast Func-hanteringen

```
class Integrator {
    Func funcObj;
      .....
    Integrator(Func fuo){
      funcObj = fuo;
    }
    void set(...){...}
    void setFunc(Func fuo){   //change function
       funcObj = fuo;
    }
    double integralValue(){
        ...
          sum += funcObj.func(x); //access
        ...                       //function
    }
}
```

Tre konkreta klasser som implementerar Func

```
class Bell implements Func{
    public double func(double x){
        return Math.exp(-x*x);
    }
}

class Sin1 implements Func{
    public double func(double x){
        return Math.sin(x)/x;
    }
}

class ArcSinh implements Func{
    public double func(double x){
        return Math.log(x+Math.sqrt(x*x+1));
    }
}
```

Hur får man integralvärden ???

```
public static void main(String[] a){
  Func bell = new Bell();
  Integrator integrator = new Integrator(bell);
  integrator.set(-0.2, 0.7, 1E-4);
  System.out.println(integrator.integralValue());

  // Byt funktionsobjekt
  integrator.setFunc(new Arcsinh());
  integrator.set(0, 0.65, 1E-3);
  System.out.println(integrator.integralValue());
}
```

Anonym inre klass för Func-objekt

```
integrator.setFunc(new Func(){
    public double func(double x){
        return (x*x+37)/(1+x);
    }
});

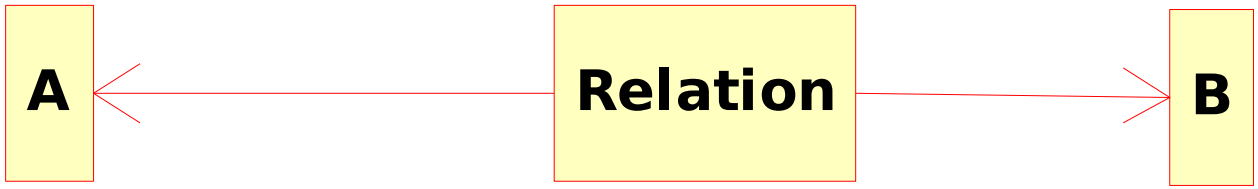
integrator.set(-0.9,0,0.01);
double val = integrator.integralValue();
```

Mönstret Relation

- A och B är associerade
- Något av följande gäller:
  - Osäkert om *A har B* eller *B har A*
  - A och B bör förbli oberoende av varandra
- Då är det lämpligt att införa ett

  relationsobjekt

Exempel på Relation



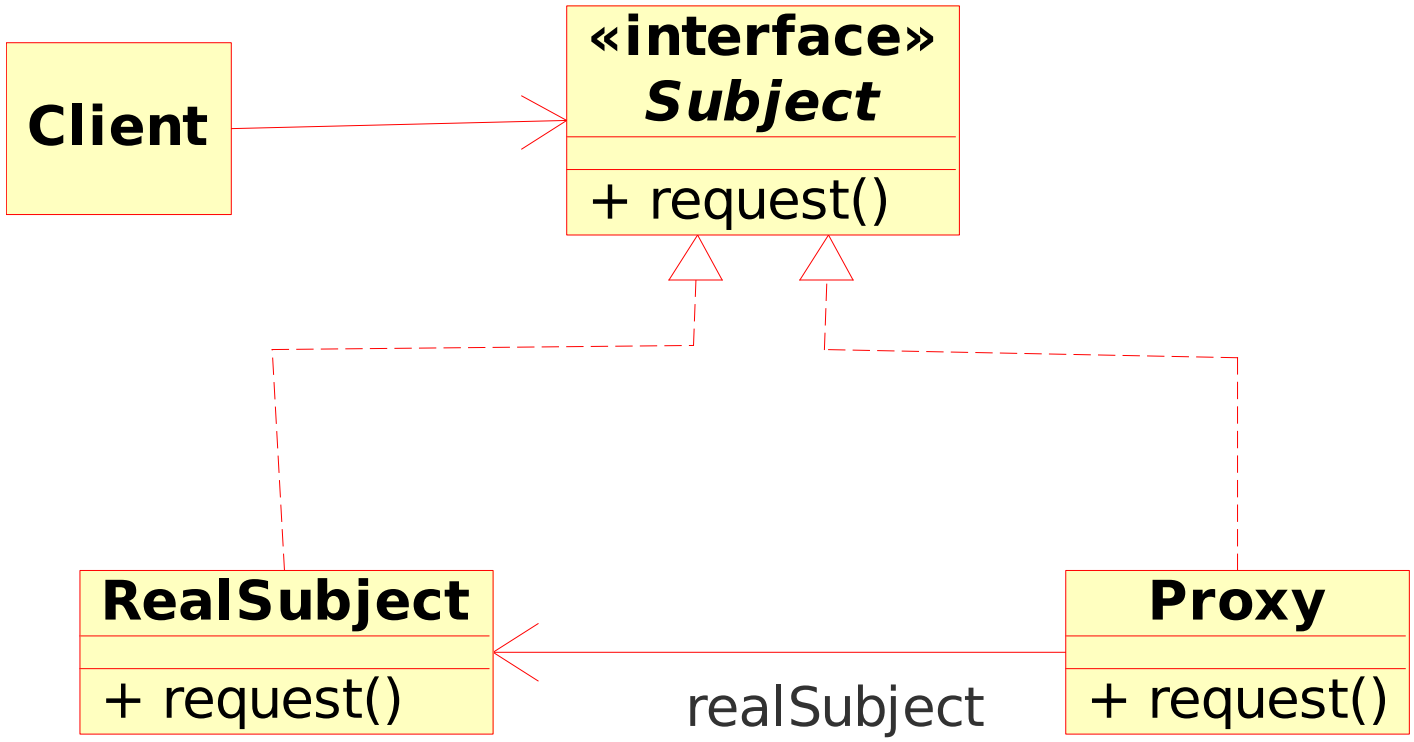
Mönstret Proxy

Proxy = ställföreträdare

En Proxy ersätter ett riktigt objekt för att

- skydda det riktiga objektet
- ”hålla ställningarna” medan
  - det riktiga objektet skapas
  - data hämtas in
  - en uppkoppling görs
- man ska slippa skapa det riktiga objektet om det inte behövs

Proxy


Client
«interface»
Subject
+ request()
RealSubject
+ request()
Proxy
+ request()
realSubject

Miniexempel på Proxy

```
interface Subject {
  public void skriv(String s);
}

class RealSubject implements Subject{
  String data;
  public void skriv(String s) { ... }
}

class Proxy implements Subject{
  RealSubject realSubject;
  //constructor necessary
  public void skriv(String s){ ... }
}
```

Miniexempel på Proxy

```
class RealSubject implements Subject{
  String data;
  public void skriv(String s) {
    data = s;
  }
}
```

Proxyn skyddar data-attributet

```
class Proxy implements Subject{
  RealSubject realSubject;
  //constructor necessary
  public void skriv(String s){
    if (skrivOK())          //must be defined !
      realSubject.skriv(s);
  }
  ...
}
```