DD2385

Programutvecklingsteknik

Några bilder till föreläsning 9

5/5 2014

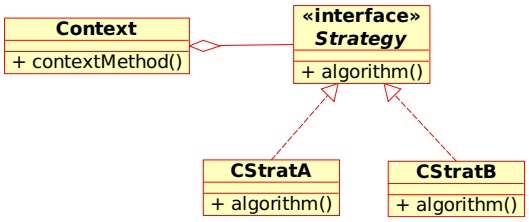
Innehåll

- Designmönster:
  - Strategy
  - Relation
  - Proxy

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

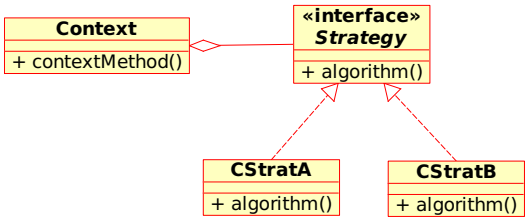
- Sortera en lista
- Objekten behöver inte vara `Comparable`
  går det? JA
- `Collections.sort(list, coll)`

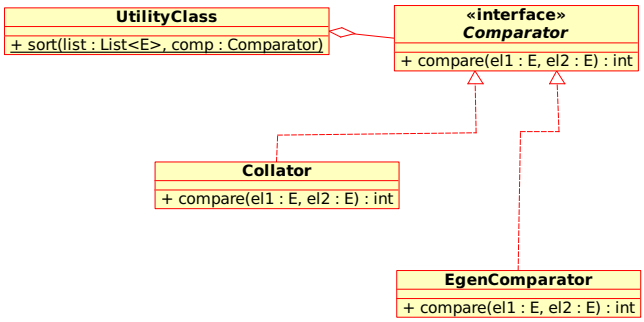
`Collections.sort(list, coll)`

- 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





Collator är en bibliotheksklass 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:

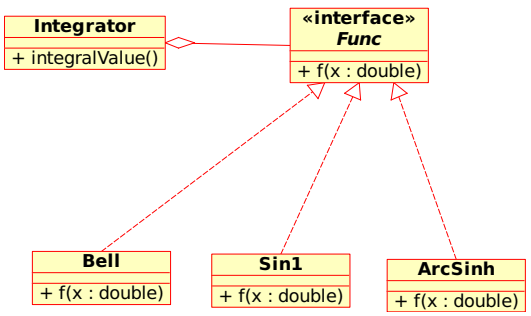
Egen `Comparator` för namnjämförelse:

```
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

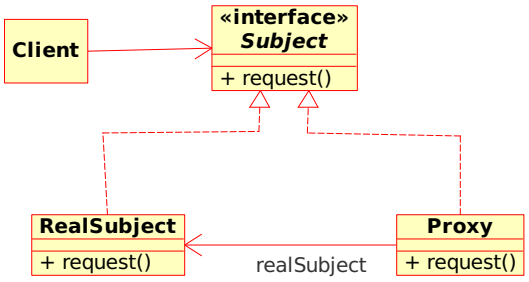
Mönstret Proxy

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



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);
  }
  ...
}
```