DD2385

Programutvecklingsteknik

Några bilder till föreläsning 7

29/4 2013

Innehåll

- Mål för programutveckling
- Designkriterier, del I
- Designmönster:
  - Template Method
  - Composite
  - Factory-teknik

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** !

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);
    }
```

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.

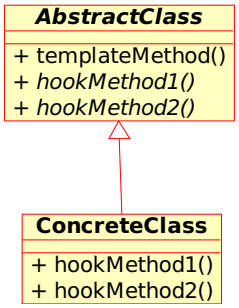
Lös koppling

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

Designmönster: Template Method

Skjut upp delar av en algoritm till subklasser

Template Method

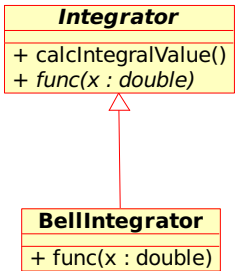


`templateMethod()`
beskriver en konkret algoritm

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());
  }
}
```

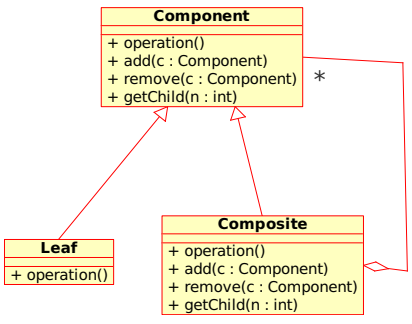
Template Method

Exempel med animering demonstreras

Designmönster: Composite

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

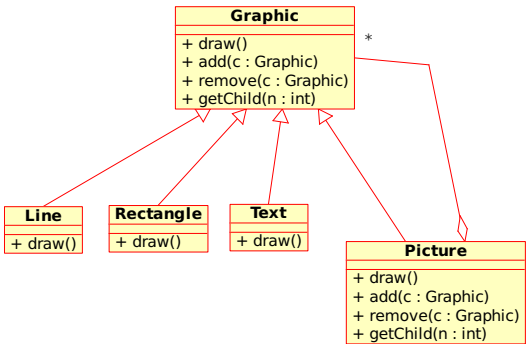
Composite



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;
}
```