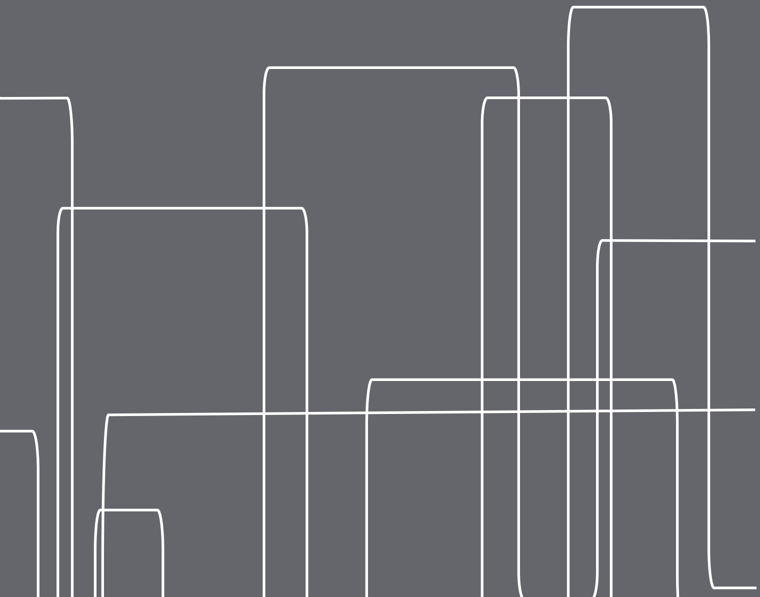
Objektorienterad Programkonstruktion

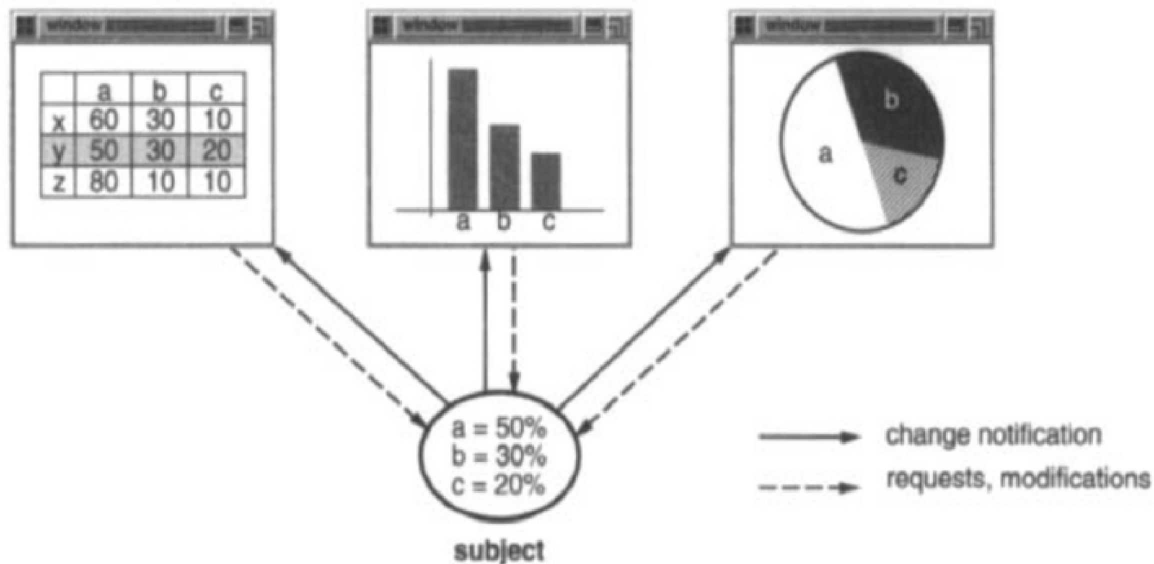
Föreläsning 7

24 nov 2015



Observer (GoF)

- Man definierar ett "ett-till-många"-förhållande mellan objekt så att när ett objekt byter tillstånd så uppdateras alla beroende objekt automatiskt
- Man slipper att explicit uppdatera alla inblandade objekt
- Tillåter lösare koppling mellan objekt, så att dessa lättare kan återanvändas
- exempel: flera View-objekt observerar ett Model-objekt

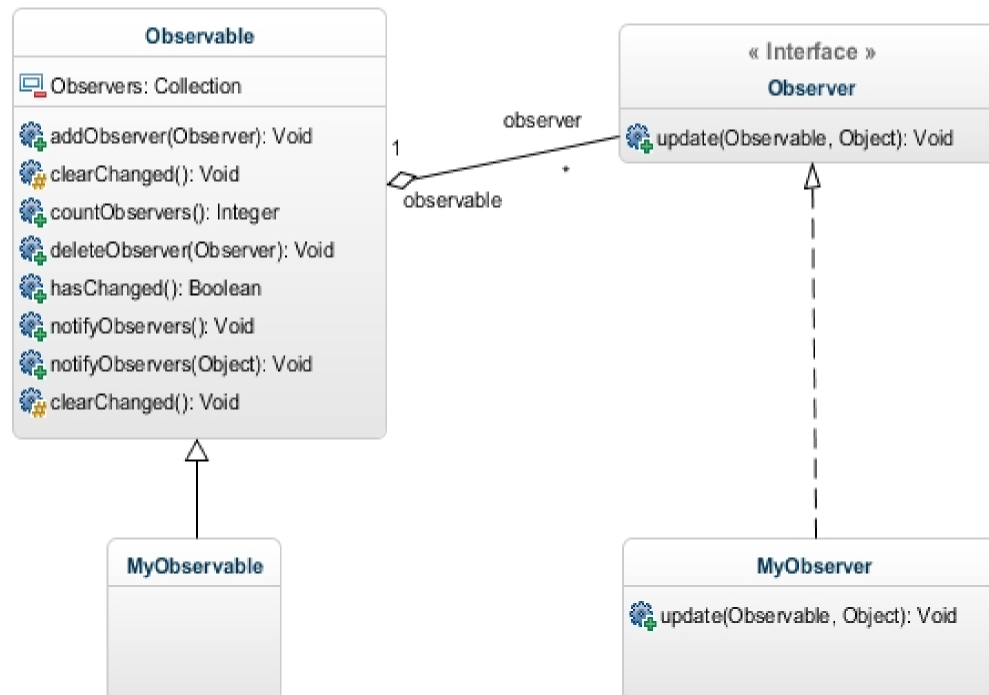


Observer

- *Behaviorial Pattern*
- Objekt **A** (**subjekt**) kan registrera ett godtyckligt antal **Observer**-objekt **B**, **C**, **D**, ..., osv
- När subjektet **A** byter tillstånd så meddelas (eng **notify**) samtliga observerobjekt **B**, **C**, **D**, ..., utan att dessa behöver känna till varandra, eller subjektet
- Javas lyssnarklasser är ett exempel på Observer

Observer & Observable

- Java har även en basklass `Observable` för subjekt, och gränsnittet `Observer` som innehåller grundläggande funktionalitet för att göra egna observatörer



Saker att tänka på

- Vilket objekt startar uppdateringen?
- Alla metoder i subjektet anropar `notify()`-metoden, som sedan meddelar alla observatörer.
  - Fördelen är att man minimerar risken för fel
  - nackdelen är att man riskerar att anropa `notify()` väldigt många gånger om man anropar flera tillståndsändringar på rad.
- Den som ändrar ett tillstånd i subjektet anropar sedan subjektets `notify()`-metod, som sedan meddelar alla observatörer.
  - Fördelen är att man kan avsluta en lång rad med tillståndsändringar med en enda uppdatering
  - Nackdelen är att man riskerar att missa anropet av `notify()`

Saker att tänka på

- Vilken information ska finnas med när subjektet meddelar sina observatörer?
- **Push** - subjektet meddelar allt som har ändrats när den meddelar sina observatörer.
  - Fördelen är att observatören vet om den behöver göra något åt saken eller inte
  - nackdelen är att vissa observatörer förses med mycket mer information än de behöver
- **Pull** - subjektet meddelar bara att något har ändrats och låter observatörerna lista ut vad.
  - Fördelen är att observatören själv kan kolla upp bara just de delar den är intresserad av
  - nackdelen är att man riskerar att många saker som inte ändrats undersöks av observatören

Publisher / Subscriber

- Alternativ till Observer
- Man definierar ett antal **kanaler** (eng. **topics**), till vilka olika objekt kan publicera information
- De objekt som prenumererar på en kanal får då all information som publiceras
- De som publicerar behöver ingen explicit information om vilka objekt (om några) som lyssnar

Factory (GoF)

- *Creational Pattern*
- I stället för att skapa objekt med en publik konstruktor så har man en eller flera publika factory-metoder som returnerar objekt av den önskade typen
- Fabriksmetoden kan inkludera kod som bedömmer vilken klass av objekt som ska returneras
- Fabriksmetoden kan bestämma om ett nytt objekt alls ska instansieras, eller om man ska returnera en referens till ett som redan finns (jmf Singleton)
- Man kan ha flera fabriksmetoder med samma argumentsignatur

Factory

- Man kan låta fabriksmetoden returnera olika klasser beroende på indata
- Exempel:
  - `LjudInspelning` och `FilmSnutt` ärver från typen `Media`.
  - metoden `mediaFactory` returnerar antingen en `LjudInspelning` eller en `FilmSnutt` beroende på vad filen `filnamn.mp3` innehåller

```
Media myMedia = mediaFactory("filnamn.mp3");
```

Factory, punktexempel

- Antag att vi har skapat en klass för punkter, `Point`
- Internt lagras punkten med en x-koordinat och en y-koordinat
- Vi vill kunna skapa punkten utifrån både kartesiska och polära koordinater
- Detta går **inte**:

```
double x = 20; double y = 20;
double r = 14; double theta = math.PI/4;
Point punktA = new Point(x, y);
Point punktB = new Point(r,theta);
```

- Javakompilatorn vet ju inte vilken konsrtuktor vi menar i de olika fallen - de har ju samma signatur!

Factory, punktexempel

```
public class Point {

    private double x, y;

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

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

Builder (GoF)

- *Creational Pattern*
- Man separerar ett objekts representation och dess konstruktion
- En Builder är ett objekt som kan sätta ihop andra objekt, steg för steg
- I labb 4 förekommer ett exempel, StringBuilder.
- Det objekt som anropar de olika stegen i Builder-objektet brukar kallas för Director (regissör)

Builder

- Man kan konstruera ett objekt i lugn och ro, utan att vara begränsad av objektets interna representation, och när man är nöjd skapas ett objekt av den önskade typen
- Många klasser som har en tillhörande Builder-klass har en konstruktor som tar ett Builder-objekt som argument.

  t.ex: `public String(StringBuilder sb);`

- Man kan ha en struktur i det färdiga objektet som är mer effektiv, och en struktur i Buildern som är mer flexibel.
- För att konstruera ett lite mer komplext objekt kan Director-objektet anropa ett stort antal olika metoder i Buildern, men man riskerar inte att någon annan tråd kommer åt ett halvfärdigt objekt innan man är klar
- En Builder kan ta ett färdigt objekt som argument i konstruktorn och använda det för att bygga ett nytt