Objektorienterad Programkonstruktion

Föreläsning 6

21 nov 2016

Designmönster

- Färdiga "recept" för att lösa (del-)problem i strukturering av ens program
- Mönster kan beskriva små komponenter eller stora strukturer
- Idéen kommer ursprungligen från arkitekturvärlden på 70- och 80-talet, där man använde funktionella beskrivningar av olika komponenter för att sätta ihop mönsterlösningar
- Formulerades för datalogin 1994 i boken "Design Patterns", av Gamma, Helm, Johnson och Vlissides (GoF)
- Har sedan utvecklats och blivit en genomgående tema i framför allt objektorienterad programmering och "software engineering"

- Exempel på hur man inte bör göra kallas "Anti-Patterns" efter en bok med samma namn av Brown et al.

Argument för och emot

- Det finns färdiga lösningar på många problem – man slipper återuppfinna hjulet
- Det är lättare att diskutera och förstå program när man har en gemensam terminologi för strukturer

- Designmönster kan leda till stel programmering, där man forcerar en dålig struktur på sin program bara för att det inte fanns någon bättre att tillgå i listan över färdiga mönster
- Vissa (t.ex LISP-programmerare som Paul Graham) hävdar att designmönster bara täcker över brister i språket, och inte behövs i bra programmeringspråk
- Svårt att återanvända mönster direkt i koden, måste ofta implementeras på nytt

Olika sorters mönster

- **Creational Patterns** - beskriver olika sätt att skapa objekt på
- **Structural Patterns** - beskriver olika sätt att realisera relationer mellan objekt
- **Behaviorial Patterns** -beskriver olika sätt att implementera kommunikation mellan objekt
- **Architectural Patterns** - beskriver olika sätt att strukturera program
- **Concurrency Patterns** - beskriver olika sätt att hantera samtidighet och turordningar i flertrådade program

Singleton

- *Creational Pattern*
- Ett objekt som det bara finns (kan finnas) en enda instans av, ex filsystem, fönsterhanterare, kontrollobjekt (se **MVC**)
- Gör det lättare att göra rätt genom bra design

Singleton

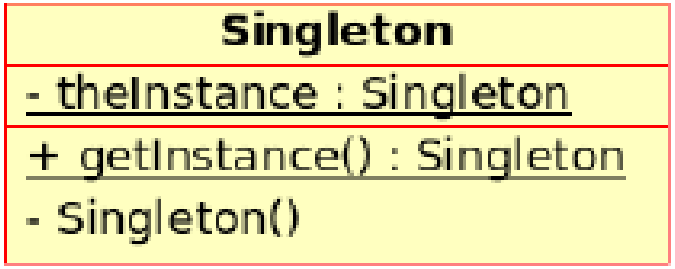
- *Creational Pattern*
- Ett objekt som det bara finns (kan finnas) en enda instans av, ex filsystem, fönsterhanterare, kontrollobjekt (se **MVC**)
- Gör det lättare att göra rätt genom bra design
- Dålig lösning: skriv i dokumentationen att man bara får skapa ett objekt av denna klass en gång

Singleton

- *Creational Pattern*
- Ett objekt som det bara finns (kan finnas) en enda instans av, ex filsystem, fönsterhanterare, kontrollobjekt (se **MVC**)
- Gör det lättare att göra rätt genom bra design
- Dålig lösning: skriv i dokumentationen att man bara får skapa ett objekt av denna klass en gång
- Bättre lösning: Gör så att det bara går att skapa ett objekt, försök att skapa fler går inte ens att kompilera
- Gör det lättare att göra rätt genom att göra det omöjligt att göra fel!

Singleton

- Man kan hindra användaren från att skapa fler objekt genom att göra konstruktorn privat. Nu får man inte längre anropa new MySingleton()
- Skapa i stället en publik metod som returnerar det enda objektet som kan finnas



Singleton i Java, version 1

```
public class MySingleton {

    private static MySingleton theInstance = new MySingleton();

    private MySingleton() {
        ...
    }

    public static MySingleton getInstance() {
        return theInstance;
    }
}
```

Singleton i Java, version 2

```
public class MyLazySingleton {

    private static MyLazySingleton theInstance = null;

    private MyLazySingleton() {
        ...
    }

    public static MyLazySingleton getInstance() {
        if (theInstance == null)
            theInstance = new MyLazySingleton();
        return theInstance;
    }
}
```

Singleton

- Det finns inget explicit hinder mot att ärva från en singleton-klass om man inte gör konstruktorn privat
- Det går att överskugga de egenskaper som gör klassen till en singleton, så ärvande klasser ärver inte singletonegenskapen
- Om man inte vill att det ska gå att ärva från ens singletonklass, kan det vara bra att deklarera klassen som `final` för att förhindra förvirrande arv

Eget arv av singleton

- Om man vill ha egna subklasser som är singleton, med den ytterligare egenskapen att det bara får finnas en instans totalt, kan man använda sig av ett paket
- Man skriver en abstrakt publik klass som har en metod för att returnera ett och endast ett objekt, men vid skapandet av objektet väljs vilken sort det skall vara
- De klasser som ärver är inte publika, och kan inte instansieras utifrån paketet
- Den abstrakta klassen är synlig utifrån, men går inte att instansiera eftersom den är abstrakt


```
package singleton;
public abstract class Singleton {

    private static Singleton theInstance;

    Singleton() {}

    public static Singleton getInstance() {
        if(theInstance == null) {
         // Välj subklass och skapa objekt
        }
        return theInstance;
    }
}
```

```
package singleton;
class SingleA extends Singleton {
    SingleA () {}
}
```

```
package singleton;
class SingleB extends Singleton {
    SingleB () {}
}
```

Fortfarande ett aktivt utvecklingsområde!

F00:an Arno Lepisk talar om Singletonimplementationer på CppCon 2016:

https://www.youtube.com/watch?v=23xDn3ReH7E

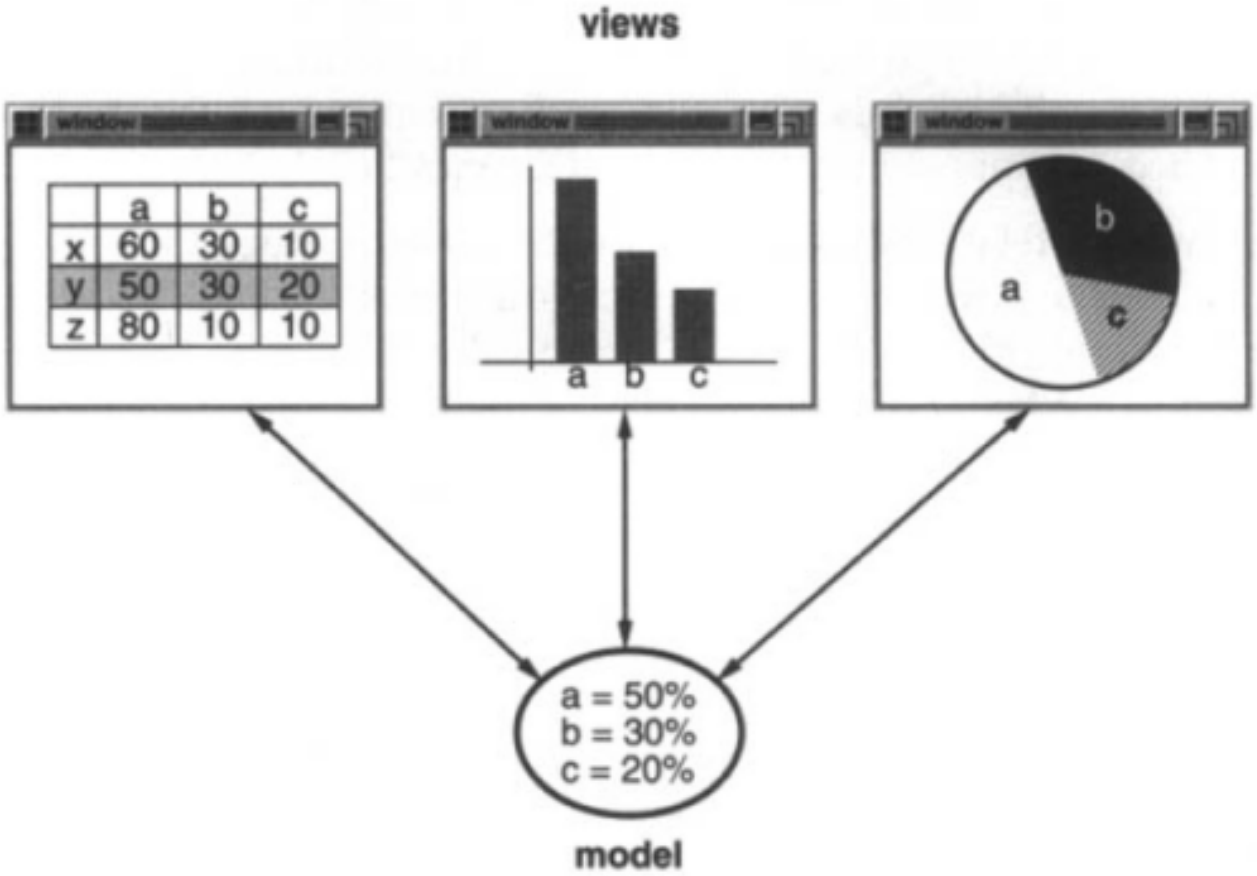
Model View Controller (MVC)

- Ett arkitekturmönster för interaktiva program
- Ett av de äldsta mönstren, (Trygve Reenskaug 1979)
- Man separerar programmet i tre funktioner
  - **Model** innehåller en model av processen, t.ex en varukorg på en e-handelsida, data och formler för en fysikalisk simulering, textdokumentet och avstavningsregler i en ordbehandlare
  - **View** presenterar resultatet för användaren, genererar t.ex hemsidan för en e-handelsida, ritar grafer för simuleringen, eller visar upp dokumentet på skärmen i ordbehandlaren
  - **Controller** styr processen, t.ex genom att säga åt modellen att uppdatera varukorgen när någon trycker på "köp"-knappen, säger åt simuleringen att börja när användaren trycker på "start", eller genom att säga åt radbrytaren att avstava texten när nya tecken skrivs in. Kan även säga åt view att uppdatera bilden

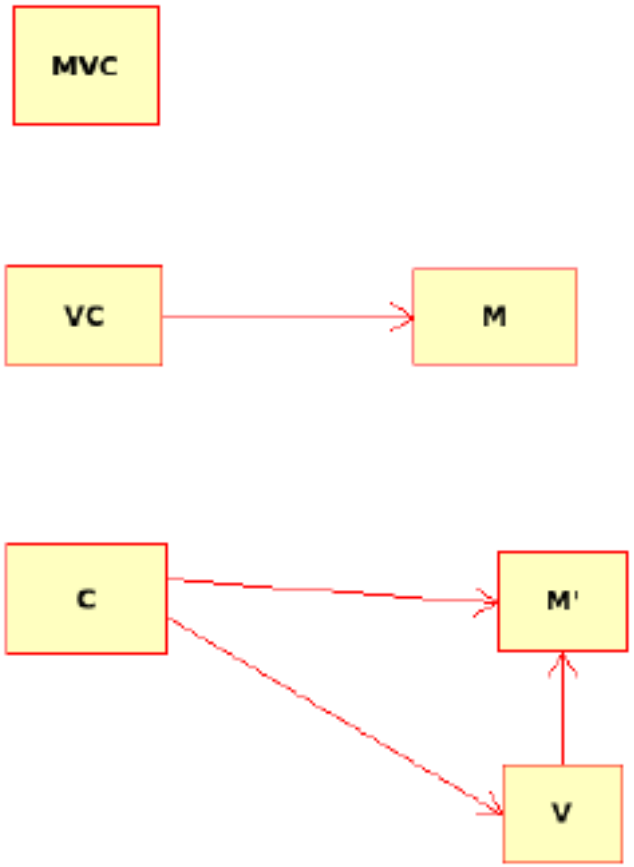
MVC

- Håller isär olika funktioner, så att det är lättare att byta ut delarna, t.ex kan man
  - byta visualisering utan att ändra något i det bakomliggande programmet
  - byta model utan att ändra visualiseringsprogrammet
- De olika delarna kan ha olika krav, så i vissa fall kan de vara separata program som körs på separata datorer
- Gör det lättare att byta platform, då platformsbyte framförallt berör view (och i viss mån controller)

MVC


views
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
b
a
c
a = 50%
b = 30%
c = 20%
model

Olika grader av updelning



MVC i Swing

- Javas Swingkomponenter är i sig ett exempel på MVC, men det är inte helt uppenbart för användaren
- Varje grafisk swingkomponent har en model, dvs ett objekt som innehåller data och metoder som inte har med grafiken att göra. Modellen är utbytbar.

- Exempel knappar:
  - I modellen för knappar finns t.ex. metoderna `addActionListener()` och `getActionCommand()`.
  - View och Control hanteras tillsammans av ButtonUI. Genom denna kan man byta det som kallas Look and Feel (L&F) på Swingkomponenter.

MVC i Swing-knappar