Properties

- Ett objekt kan ha vissa egenskaper (som beskrivs med instansvariabler).
- En enkel egenskap `X` avläses och sätts med metoderna `getX()` och `setX(värde)`.
- En indexerad egenskap (från t.ex. en array) avläses och sätts med metoderna `getX(i)` och `setX(i,värde)`
- När en enkel egenskap ändras kan objektet generera en händelse av typen `PropertyChangedEvent`.
- När en indexerad egenskap ändras kan objektet generera en händelse av typen `IndexedPropertyChangedEvent`, en subklass till `PropertyChangedEvent`.
- Exempel på detta är standarsklasserna i `Swing`.
- Händelser kan fångas upp av lyssnare av typen `PropertyChangeListener` som har en metod med namnet `propertyChanged`.

Användbara metoder som kan anropas i `propertyChanged`:

- `getPropertyName`
- `getNewValue`
- `getOldValue`
- `getIndex` (för händelser av typen `IndexedPropertyChangedEvent`)

JavaBeans

Ett standardmönster:

- Har parameterlös konstruktor
- Implementerar `Serializable`
- Har egenskaper
- Har metoder `setX`, `getX`, `isX`
- Kan ha indexerade metoder `setX`, `getX`, `isX`

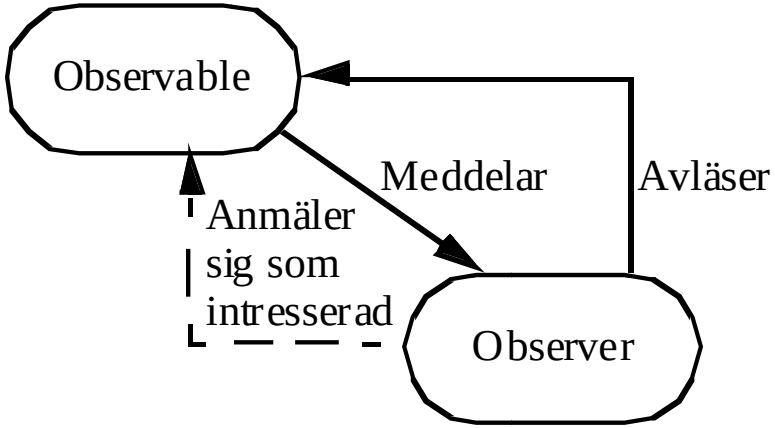
En JavaBean kan registrera lyssnare för händelser av typen `PropertyChangeListener`.
Kan t.ex. implementera följande gränssnitt:

```
import java.beans.*;
import java.io.*;
public interface MyBeanModel extends Serializable {
  void addPropertyChangeListener(PropertyChangeListener l);
  void removePropertyChangeListener(PropertyChangeListener l);
}
```

```
import java.beans.*;
public class MyBeanClass implements MyBeanModel {
  private final PropertyChangeSupport p=new PropertyChangeSupport(this);

  public void addPropertyChangeListener(PropertyChangeListener l) {
    p.addPropertyChangeListener(l);
  }
  public void removePropertyChangeListener(PropertyChangeListener l) {
    p.removePropertyChangeListener(l);
  }
  private double value;
  private double[] arr;
  public MyBeanClass () { ... }  // konstruktor
  public double getValue() { return value; }
  public void setValue (double newValue) {
    double oldValue = value;
    value = newValue;
    p.firePropertyChange("value", oldValue, newValue);
  }
  public double getArr(int index) { return arr[index]; }
  public void setArr(int index, double newElem) {
    double oldElem = arr[index];
    arr[index] = newElem;
    p.fireIndexedPropertyChange("arr", index, oldElem, newElem);
  }
}
```

Observer pattern

Ett vanligt s.k. *designmönster*



Kallas Model-View för grafiska komponenter.

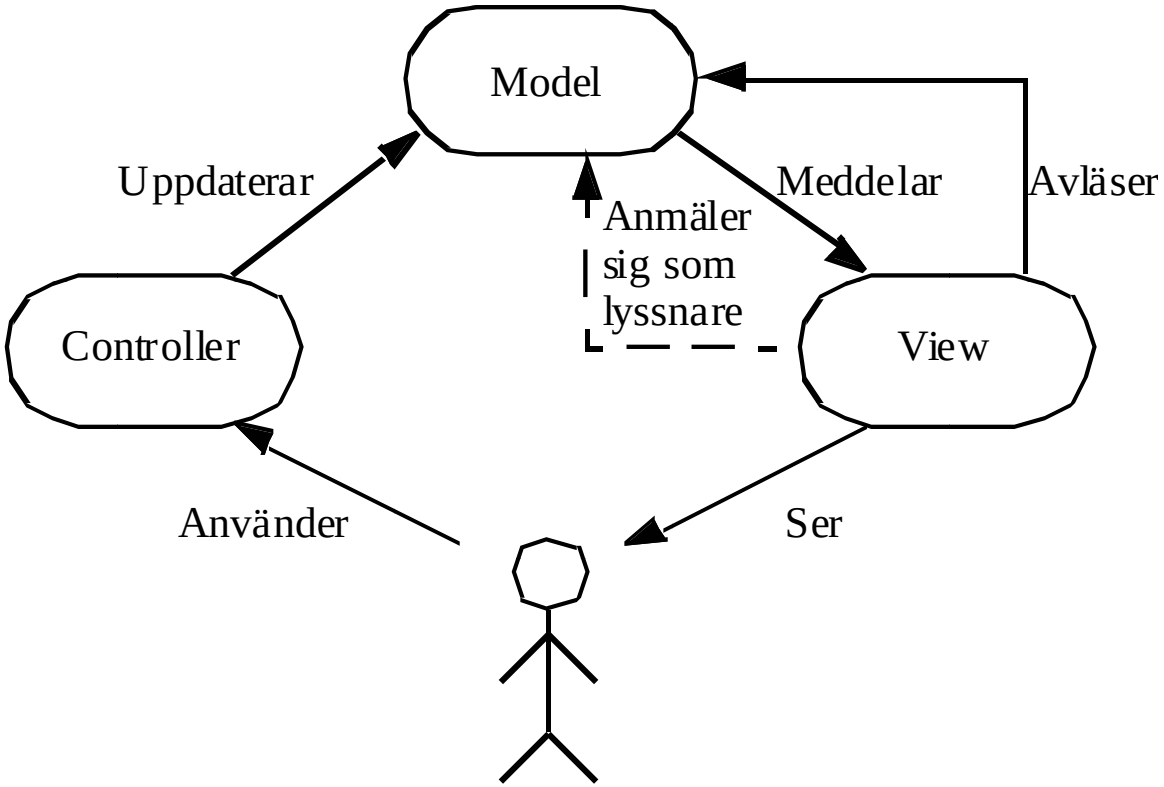
Modellen kan göras som en JavaBean.

Vyn, ett exempel:

```
import java.beans.*;
import javax.swing.*;

public class MyView extends JPanel
                            implements PropertyChangeListener {
   private MyBeanModel model;

   public MyView(MyBeanModel m) {
     model = m;
     model.addPropertyChangeListener(this);

    // utplacering av grafiska komponenter
     . . .
   }

   public void propertyChange(PropertyChangeEvent e) {
     double v = model.getValue();
     // ändra grafiska komponenter
     . . .
   }
 }
```

Model-View-Controller

Controllern tar emot signaler utifrån och uppdaterar modellen:



```
import javax.swing.*;
import java.awt.event.*;
public class MyController extends JPanel
                          implements ActionListener {
  private MyBeanModel model;
  private JTextField f = new JTextField(5);
  public MyController (MyBeanModel m) {
    model = m;
    add(f);
    f.addActionListener(this);
  }
  public void actionPerformed(ActionEvent e) {
    if (model != null)
      model.setValue(Integer.parseInt(f.getText()));
  }
}
```

Problem i praktiken:

- Komponenter i vyn används ofta även för inmatning (användaren klickar t.ex.)
- Svårt att separera Controllern från Vyn
- I Swing används s.k. UI-delegates som fungerar både som vy och controller.