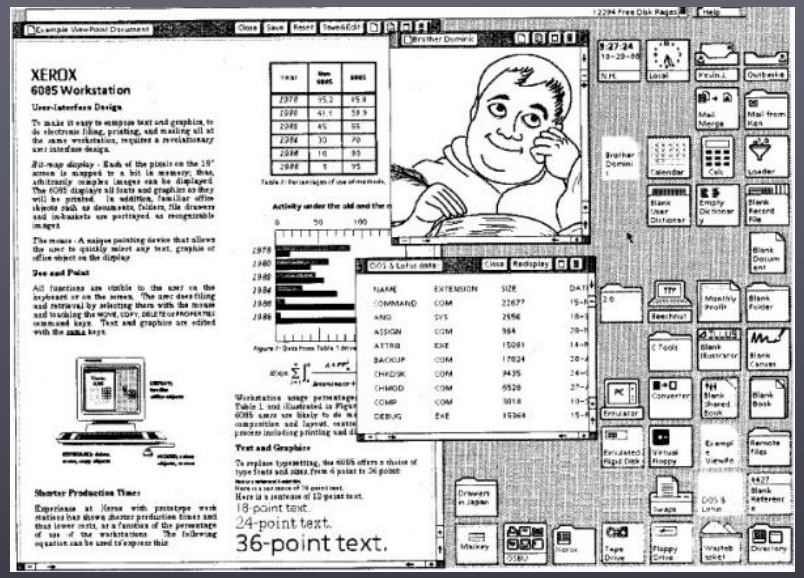



# GUI-programmering

Gustav Taxén
gustavt@csc.kth.se

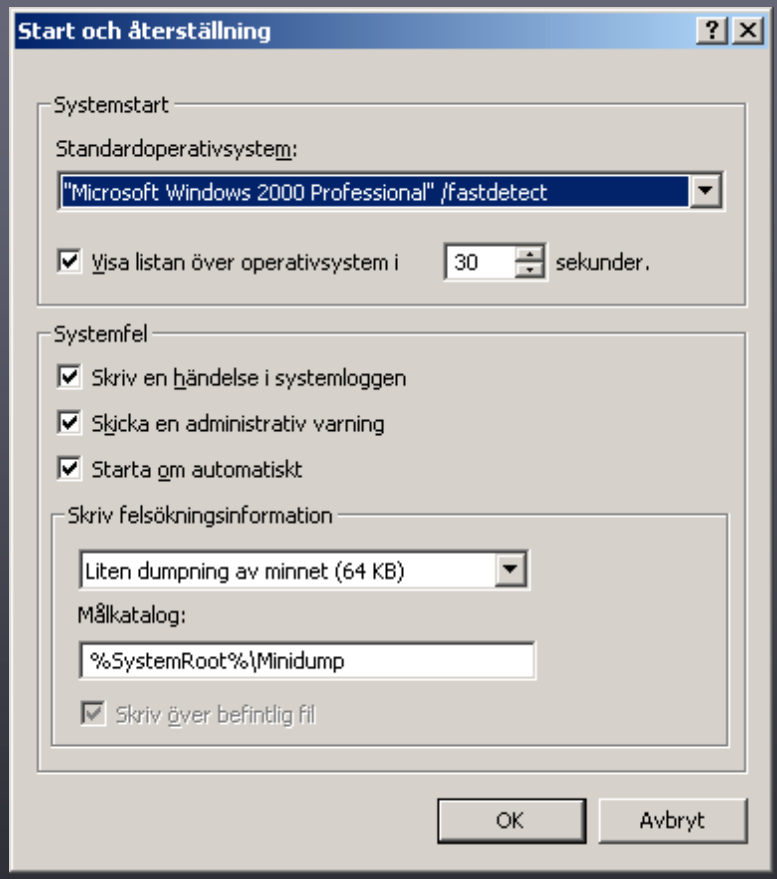
Martin Berglund
mabe02@kth.se

# WIMP

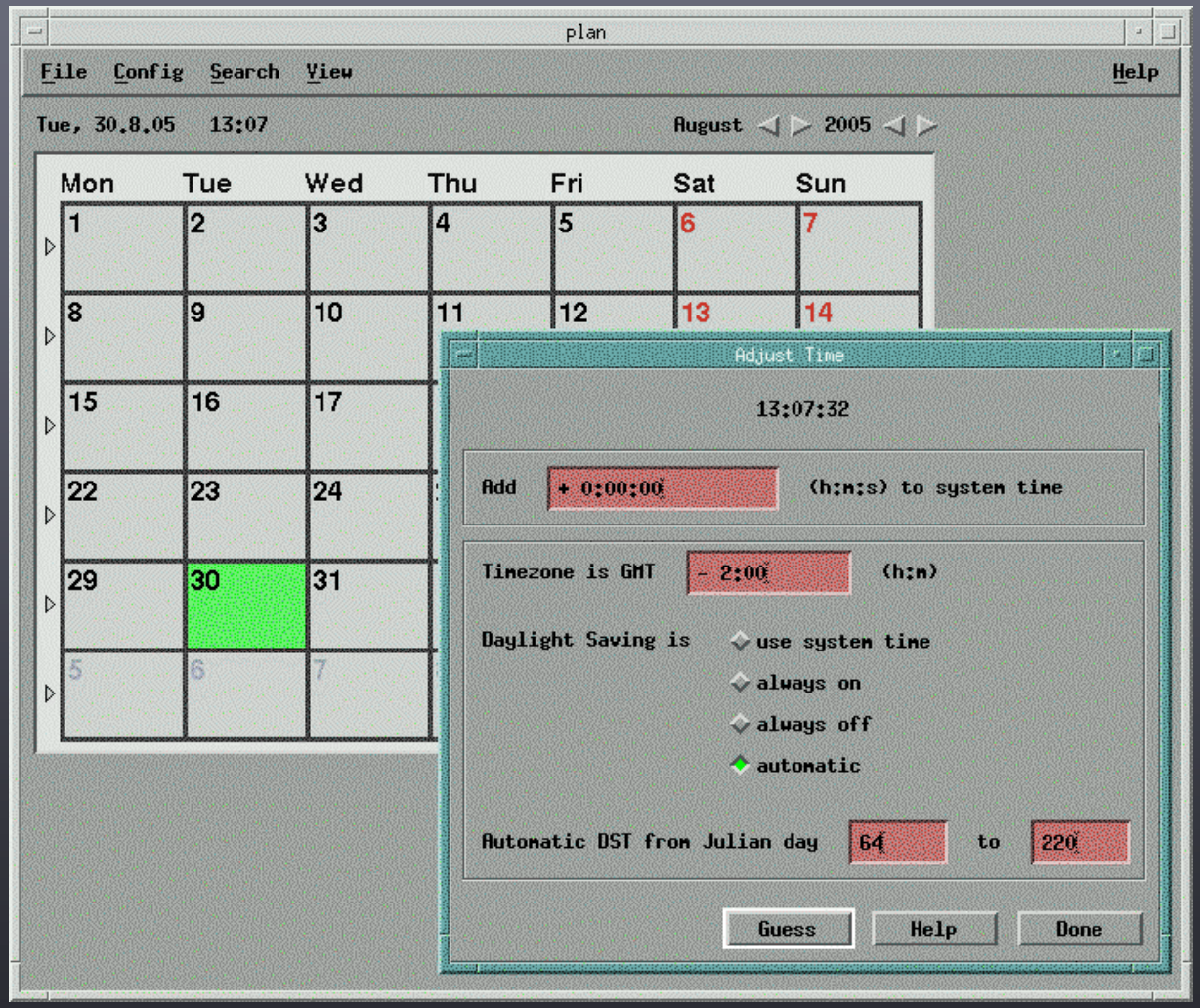
- Window
- Icon
- Menu
- Pointing device

# Modernt Gränssnitt

- Vad ingår i ett gränssnitt?
- Vad förväntar vi oss av gränssnittet?
- Bra respektive dåliga gränssnitt?
- Komponenter?
- ...och sen då?



# Modernt Gränssnitt?



# GUI till tusen

- Win32-API (C)
- MFC (C++)
- VCL/CLX (C++/Delphi)
- wxWidgets (C++)
- Qt (C++/Java)
- GTK+ (C)
- .NET Framework (VB/C++/C#)
- Java AWT/Swing/SWT (Java)
- osv...

# GUI till tusen

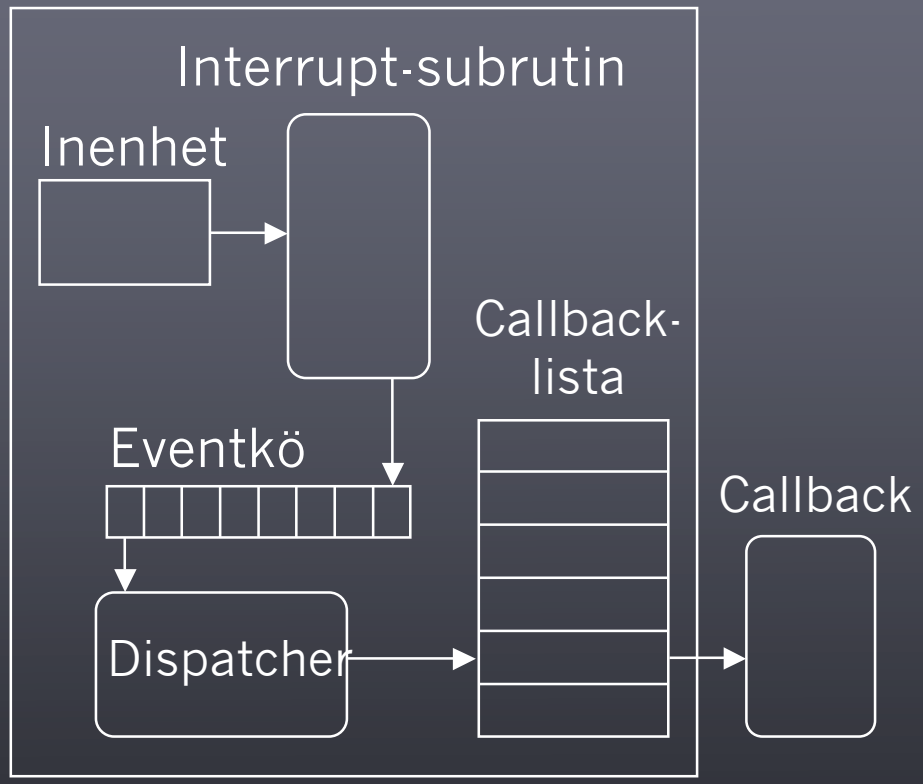
- Vi ska titta på Java Swing
- Mycket gemensamt med andra GUI-system, ibland under andra namn
- Bra uppsättning komponenter
- Platformsoberoende
- ”Bra” designat enligt designmönster
- Gratis utvecklingsmiljö (NetBeans) och mycket dokumentation från Sun

# GUI-teori

- Nu tittar vi på bakomliggande GUI-teorier
- Händelseloop i C
- Objekt-orientering i C++/Delphi
- Event-baserat i Java/C#
- Vad är bäst? Kan jag välja?

# Inmatningstyper: Events

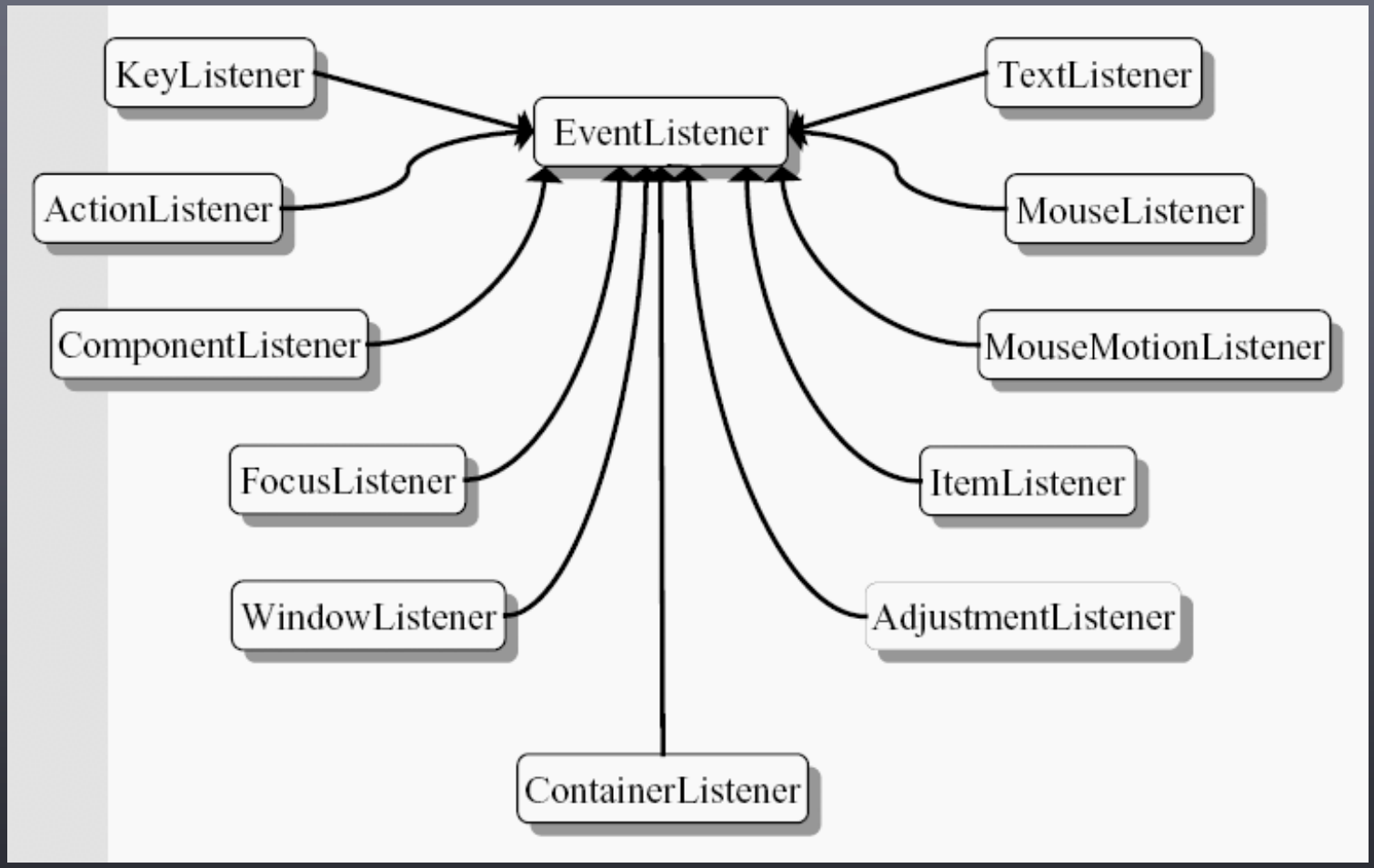
- Systemet placerar förändringar i en kö.
- Beta av kön vid lämpliga tillfällen.
- Skicka en s.k. **event-instans** med information om varje förändring till alla callbacks som är "intresserade".



# Events i Java

- Man registrerar **Listeners** som var och en "lyssnar" på en viss typ av event
- Varje metod i en listener motsvarar något som hänt hos enheten i fråga

# Events i Java



# Events i Java

```
package MyTests;
import java.awt.*;
import java.awt.event.*;
public class MyFrame1 extends Frame implements
WindowListener{
  public void windowOpened(WindowEvent e) {}
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
  public static void main(String [] args) {
    MyFrame1 frame = new MyFrame1();
    frame.addWindowListener(frame);
    frame.setVisible(true);
  }
}
```

# ...som inre klass

```
package MyTests;
import java.awt.*;
import java.awt.event.*;
class MyWindowListener implements WindowListener {
  public void windowOpened(WindowEvent e) {}
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
}

public class MyFrame2 extends Frame {
  public static void main(String [] args) {
    Frame frame = new MyFrame2();
    frame.addWindowListener(new MyWindowListener());
    frame.setVisible(true);
  }
}
```

# Adaptors

- Förenklar hanteringen av events
- Varje adaptor implementerar ett Listener-interface, men alla metoderna är tomma
- Om man subklassar en adaptor behöver man alltså bara skriva precis den kod som behövs

# Adaptors

```
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}

public class MyFrame3 extends Frame {
  public static void main(String [] args) {
    Frame frame = new MyFrame3();
    frame.addWindowListener(new MyWindowAdapter());
    frame.setVisible(true);
  }
}
```

# ...eller med anonym subklass

```
public class MyFrame4 extends Frame {
  public static void main(String [] args) {
    Frame frame = new MyFrame4();

    frame.addWindowListener(new WindowAdapter (){
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });

    frame.setVisible(true);
  }
}
```

```
public class PlayerSteeringBehaviour
             implements Behaviour, KeyListener {

  protected boolean steering = false;
  protected float direction = 1.0f;

  public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == 37) {   // Cursor left
      steering = true;
      direction = -1.0f;
    }
    if (e.getKeyCode() == 39) {   // Cursor right
      steering = true;
      direction = 1.0f;
    }
  }

  public void keyReleased(KeyEvent e) {
    steering = false;
  }

  public void keyTyped(KeyEvent e) {}

...
```

# Avancerade gränssnitt

- Om man bygger mer avancerade program behövs ett bättre sätt att hantera användargränssnittet
- De flesta system bygger på ett **designmönster** som heter **Model-View-Controller** (eller bara **Model-View**)

# Designmönster (design patterns)

- Utgår från arkitekten Christopher Alexanders arbete på 70-talet
- Alexander försökte se mönster i hur man löst återkommande problem inom arkitektur och skrev en bok där han beskriver lösningarna
- Alexander, C. (1977). **A Pattern Language**. Oxford University Press
- Idén togs upp på allvar inom systemdesign i mitten av 90-talet
- Den mest kända boken är
  Gamma et al. (1995). **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley

# Design patterns

- Namn och generell typ
- Intent (vad den gör)
- Also Known As
- Motivation
- Applicability
- Structure (ett UML-diagram)

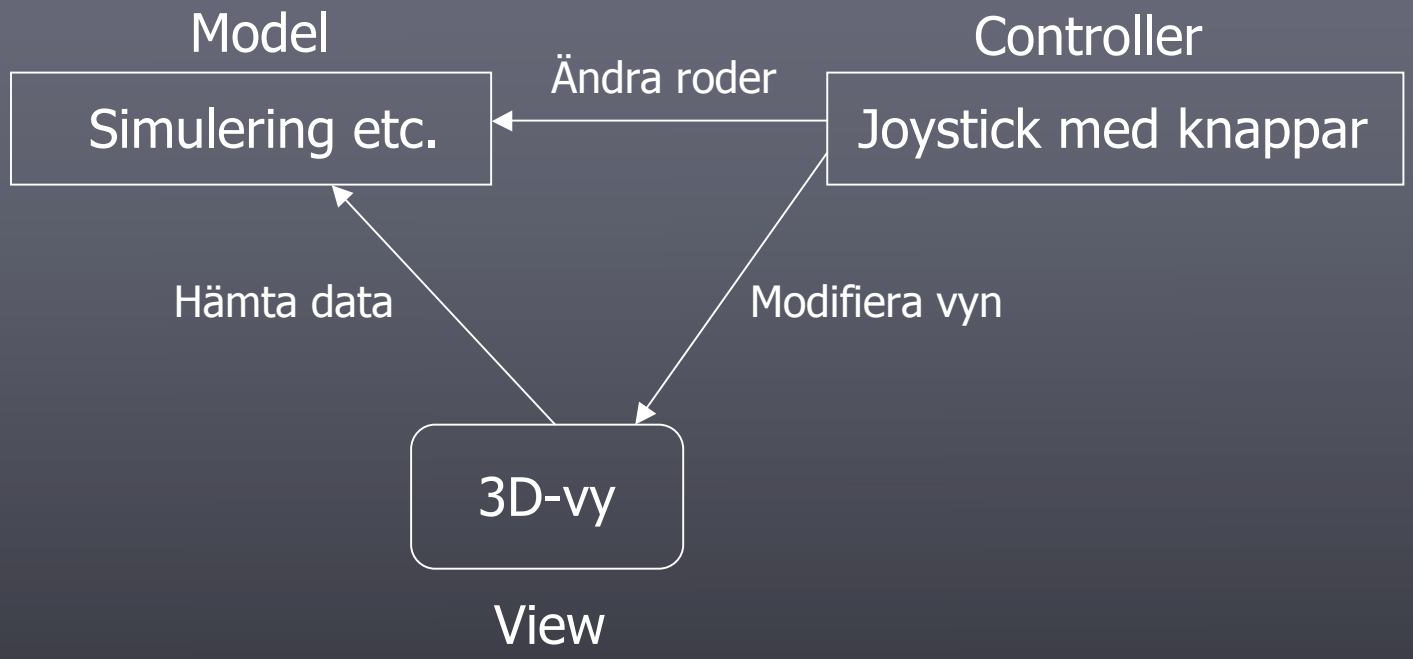
# Design patterns

- Participants (beskr. av klasser som ingår)
- Collaborations (hur klasserna arbetar tillsammans)
- Consequences (av att använda mönstret)
- Implementation
- Sample code
- Known uses
- Related patterns

# Model View Controller

- En design pattern för användargränssnitt.
- Utvecklades under sent 70-tal för Smalltalk på Xerox-maskiner
- Exempel: Flygsimulator
  - **Model** - simuleringsrutiner, etc.
  - **Controller** - joystick kopplad till datorn
  - **View** - vy genom cockpitfönstret

# Model View Controller

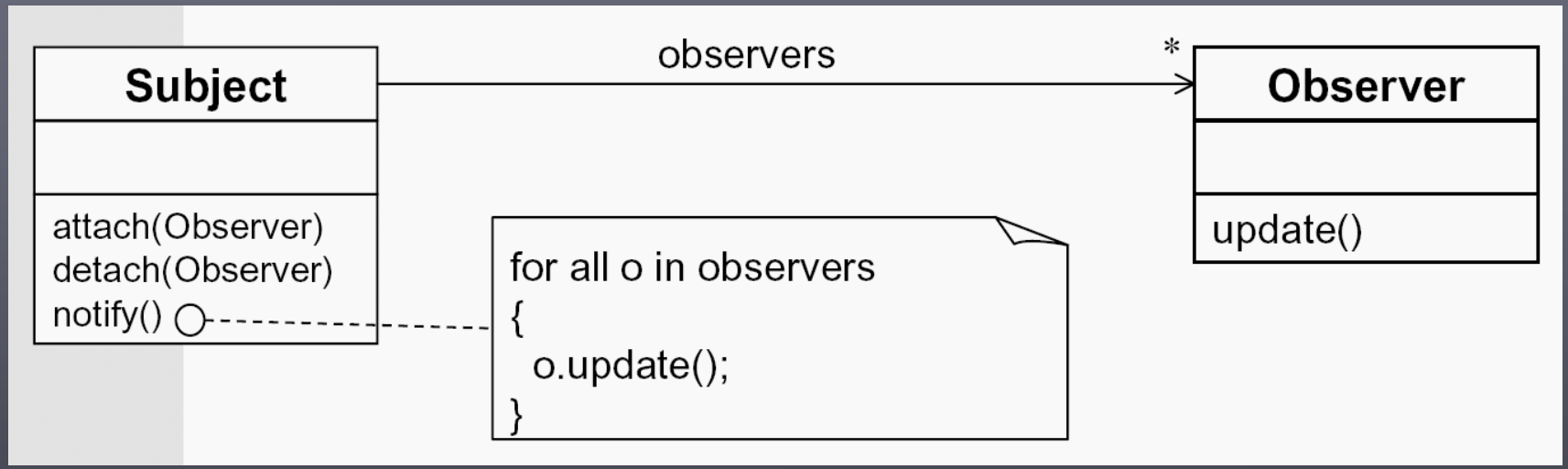



Om gränssnitten mellan de tre komponenterna inte ändras kan vilken som helst av dem bytas ut!

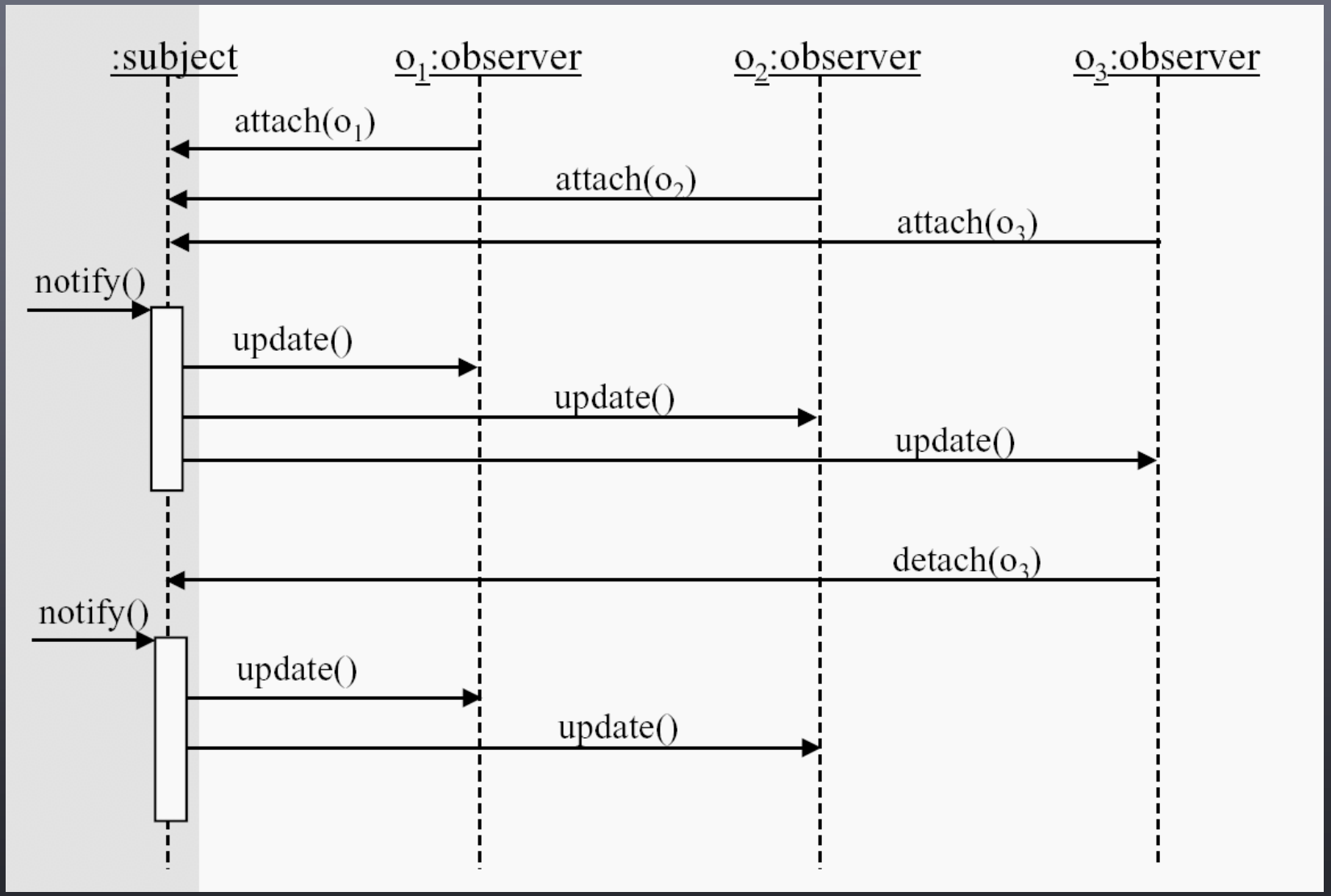
# Observer

- För att implementera MVC används ofta designmönstret **Observer**
- Tillåter att ett objekt meddelas om det gjorts en förändring i ett annat objekt, utan att objekten görs starkt knutna till varandra

# Observer


Subject
observers
*
Observer
attach(Observer)
detach(Observer)
notify()
update()
for all o in observers
{
o.update();
}

# Observer


:subject
o1:observer
o2:observer
o3:observer
attach(o1)
attach(o2)
attach(o3)
notify()
update()
update()
update()
detach(o3)
notify()
update()
update()

# Observer i Java

```
package java.util;

public interface Observer {
  void update(Observable o, Object arg);
}

public class Observable {
  private Observer[] arr;

  public void addObserver(Observer o) {
    arr.addElement(o);
  }

  public void notify(Object arg) {
    for (int i = 0; i < arr.size; i++) {
      arr[i].update(this, arg);
    }
  }
}
```

Det här är inte hela sanningen, men principen är densamma i den riktiga java-implementationen!

# Problem med MVC

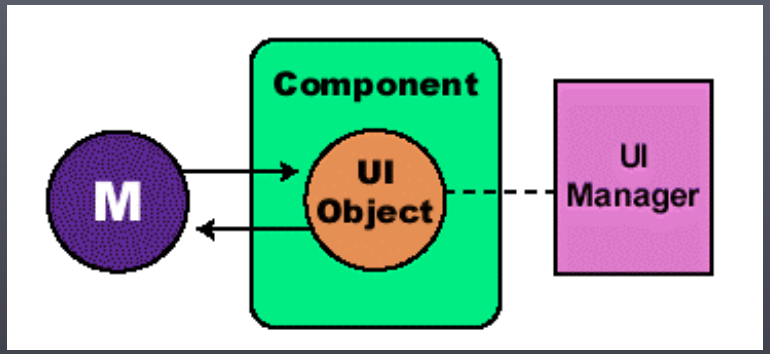
- Det kan vara svårt att separera kontroller och vy i moderna grafiksystem
- Men det lönar sig i princip alltid att separera datamodellen från gränssnittet

# Java/Swing

- Gränssnitt i Java (och de flesta andra GUI-system) är **hierarkiska**.
- Man börjar med en s.k. **Top Level Container**, och lägger till komponenter i den.
- Vissa komponenter är också containers och kan innehålla andra komponenter.
- Man specificerar dimensioner och position för varje komponent explicit eller via s.k. **Layout Managers**.

# Separable Model Architecture

- Varje komponent hanterar view/control.
- Själva utritningen är delegerad till ett s.k. UI object (för att man ska kunna ändra look-and-feel, t.ex.).
- Varje komponent har en modell kopplad till sig.



# Separable Model Architecture

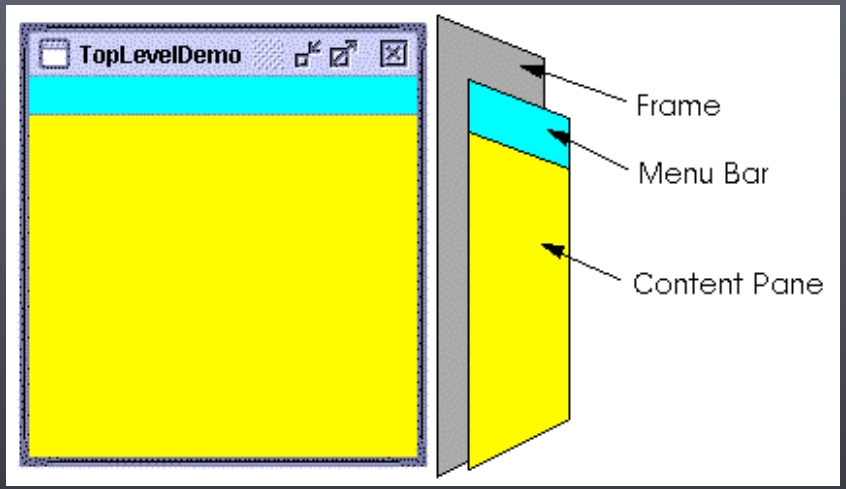
- De flesta komponenters modell har enbart att göra med själva gränssnittet (t.ex. om en checkbutton är nedtryckt eller ej).
- Vissa komponenters innehåll kan dock bara definieras av applikationen (t.ex. innehållet i en lista).
- Några faller mitt emellan (t.ex. sliders).
- Man ersätter en modell genom att subklassa defaultmodellen och sedan anropa `setModel()` på komponenten.

# Containers, Controls och Displays

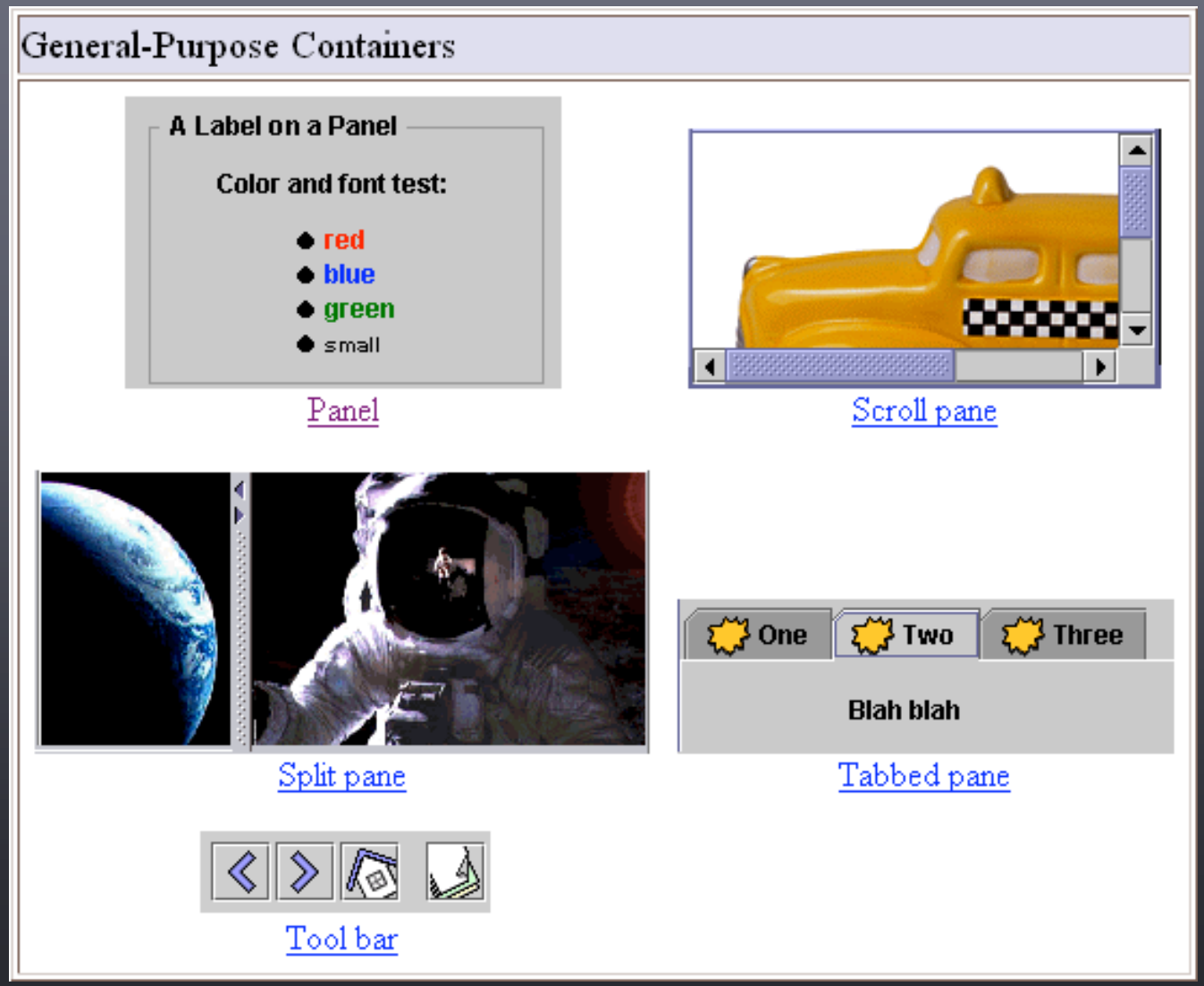
- **Top Level Containers**: Applet, Dialog, Frame.
- **General-Purpose Containers**: Panel, Scroll Pane, Split Pane, Tabbed Pane, Tool Bar.
- **Special-Purpose Containers**: Internal frame, Layered Pane, Root Pane.
- **Basic Controls**: Buttons, Lists, Menus, etc.
- **Uneditable Info Displays**: Label, Progress Bar, Tool Tip.
- **Interactive Displays of Highly Formatted Information**: File Chooser, Color Chooser, etc.

# Top Level Containers

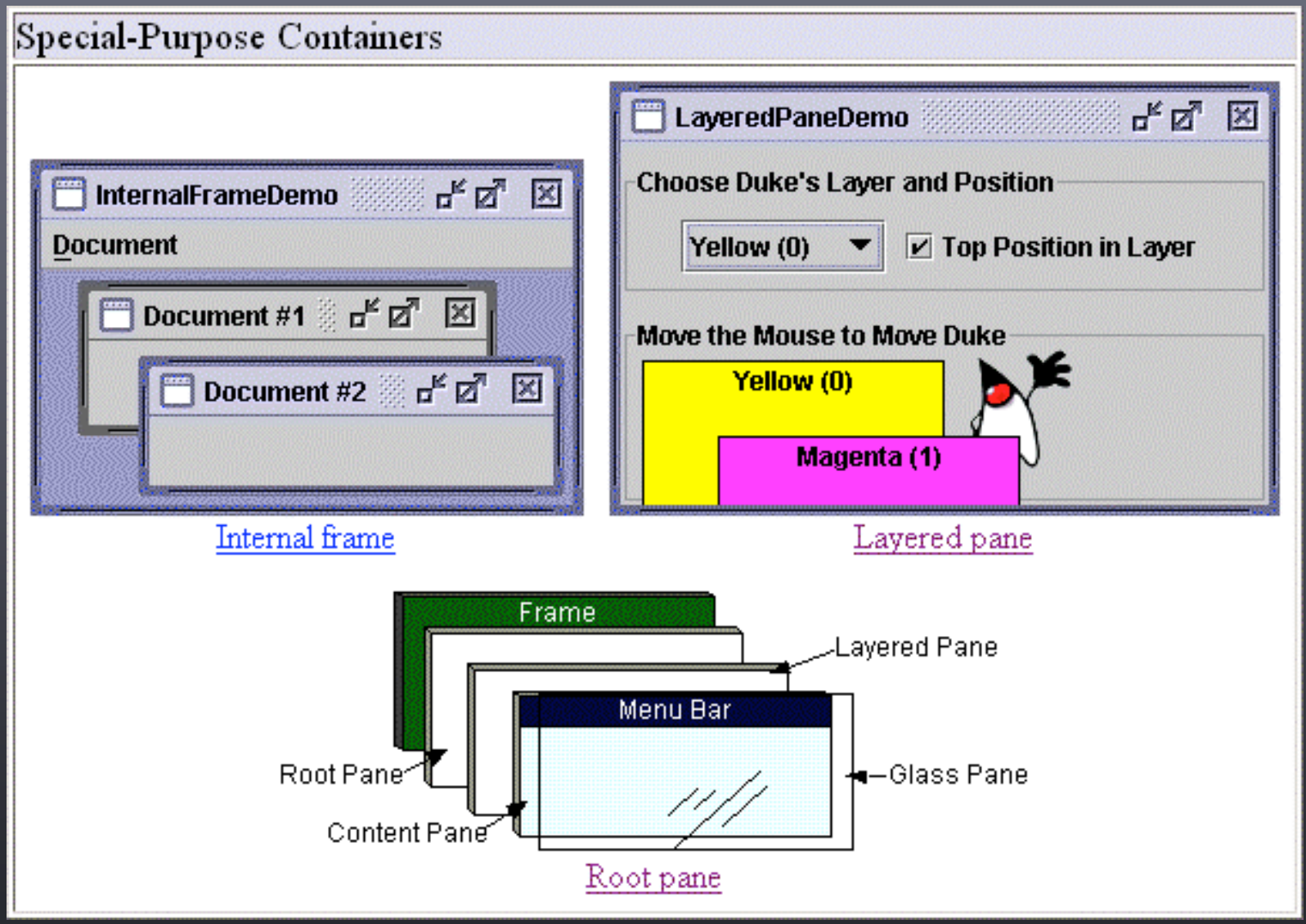
- Varje komponent kan bara befinna sig på en plats i hierarkin.
- Varje Top Level Component har en **Content Pane** som innehåller hierarkin.
- Man kan lägga till en **menu bar** (menyrad) ovanför sin content pane.



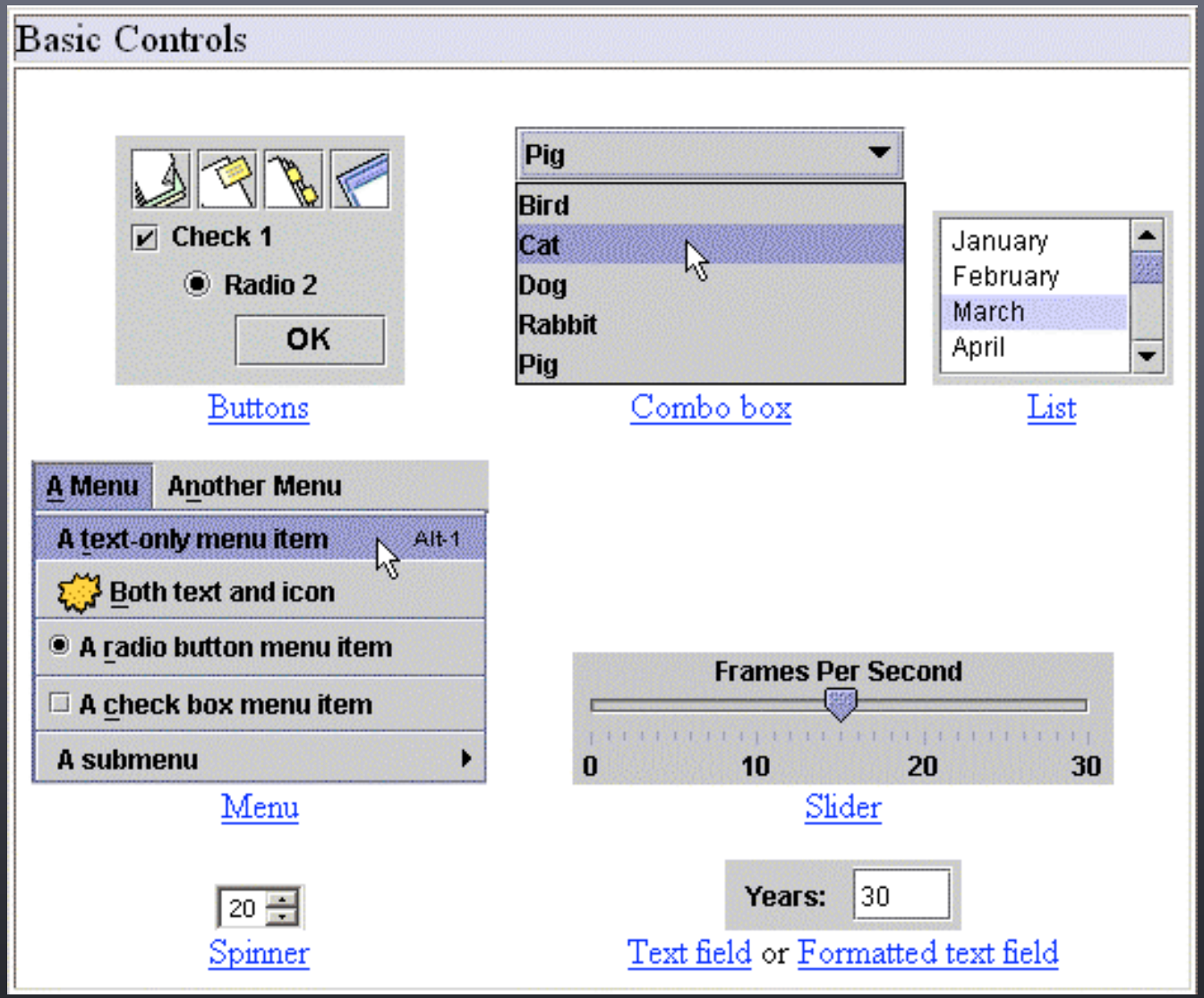
# General Purpose Containers



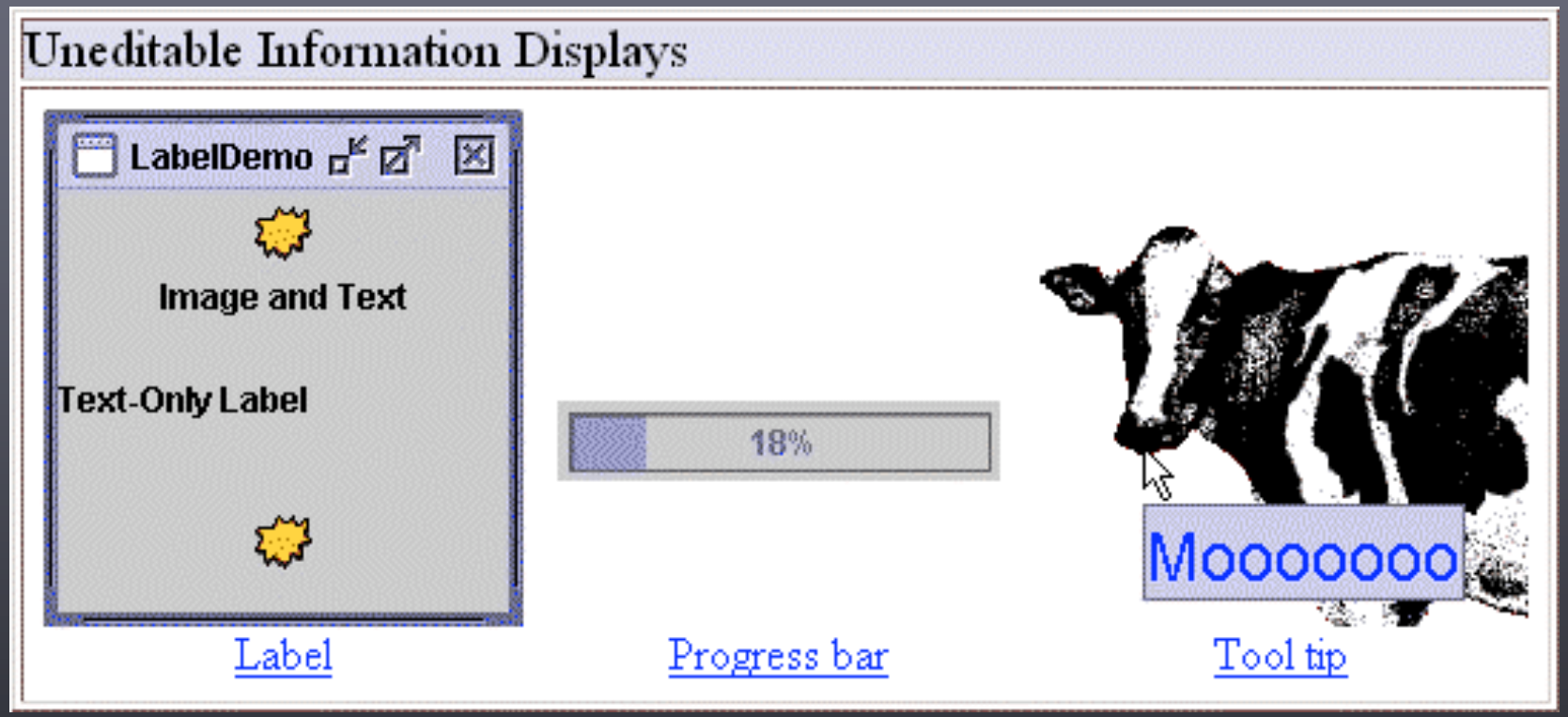
# Special Purpose Containers



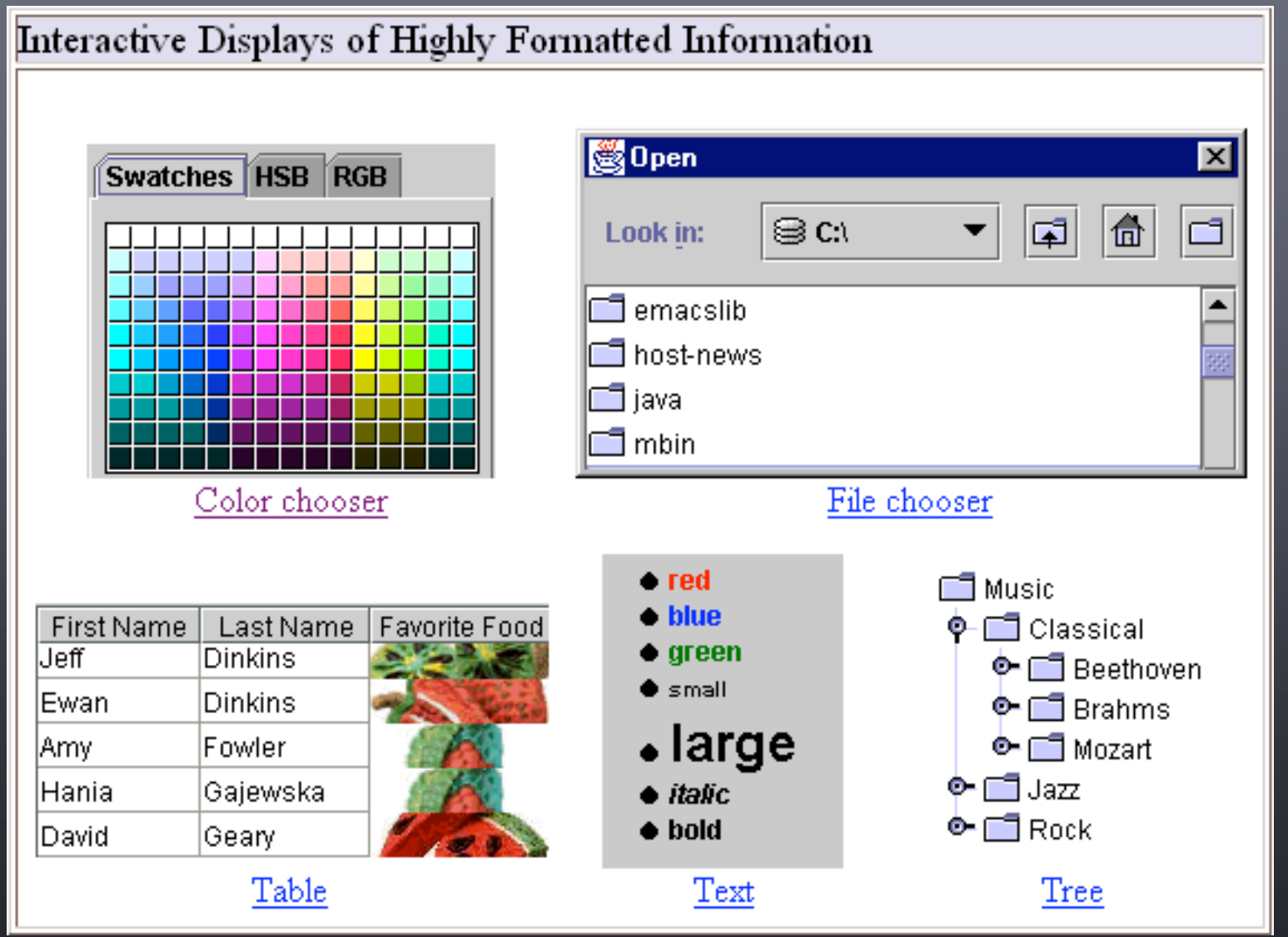
# Basic Controls



# Uneditable Info Displays



# Formatted Info Displays



# Kan vi göra något åt utseendet?

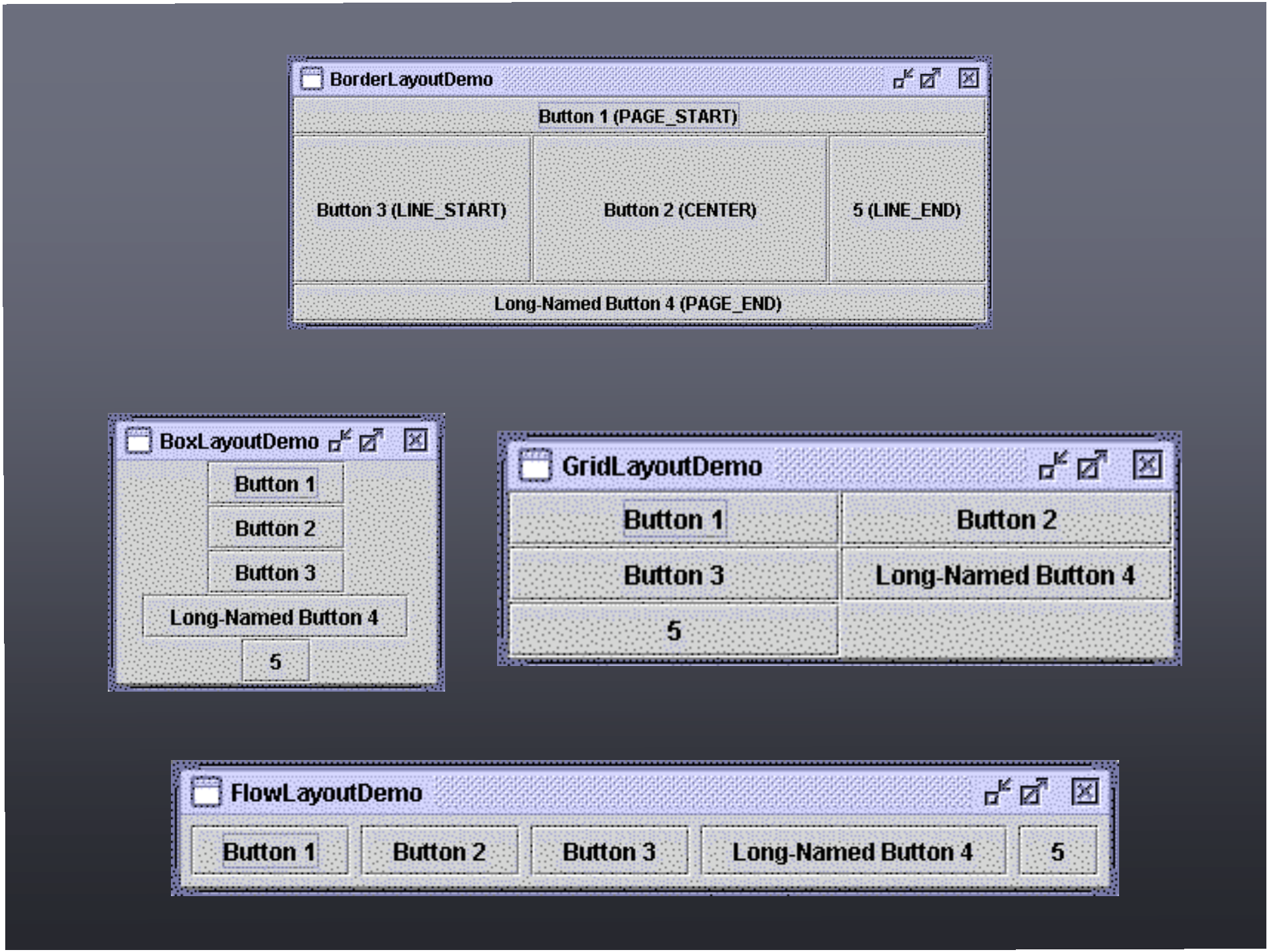
- Ja, det kan vi!
- Swing kan få ”native look & feel” om vi lägger till några rader kod…

```
// Get the native look and feel class name
String nativeLF = UIManager.getSystemLookAndFeelClassName();

// Install the look and feel
try {
        UIManager.setLookAndFeel(nativeLF);
}
catch(Exception e) {
}
```

# Layout Managers

- Det är oftast en nackdel att specificera position och dimension för komponenter explicit.
- Om användaren t.ex. ändrar fönstrets storlek kommer inte storleken på komponenterna att hänga med.
- Layout managers hjälper en att räkna ut positioner och dimensioner dynamiskt!
- Man kan skapa egna layout managers.


BorderLayoutDemo
Button 1 (PAGE_START)
Button 3 (LINE_START)
Button 2 (CENTER)
5 (LINE_END)
Long-Named Button 4 (PAGE_END)
BoxLayoutDemo
Button 1
Button 2
Button 3
Long-Named Button 4
5
GridLayoutDemo
Button 1
Button 2
Button 3
Long-Named Button 4
5
FlowLayoutDemo
Button 1
Button 2
Button 3
Long-Named Button 4
5

# Händelsehantering


```
public class Beeper ... implements ActionListener {
   ...

     // where initialization occurs:
     // create the button first, then:
     button.addActionListener(this);

   ...

   public void actionPerformed(ActionEvent e) {
     // Make a beep sound...
   }
}
```

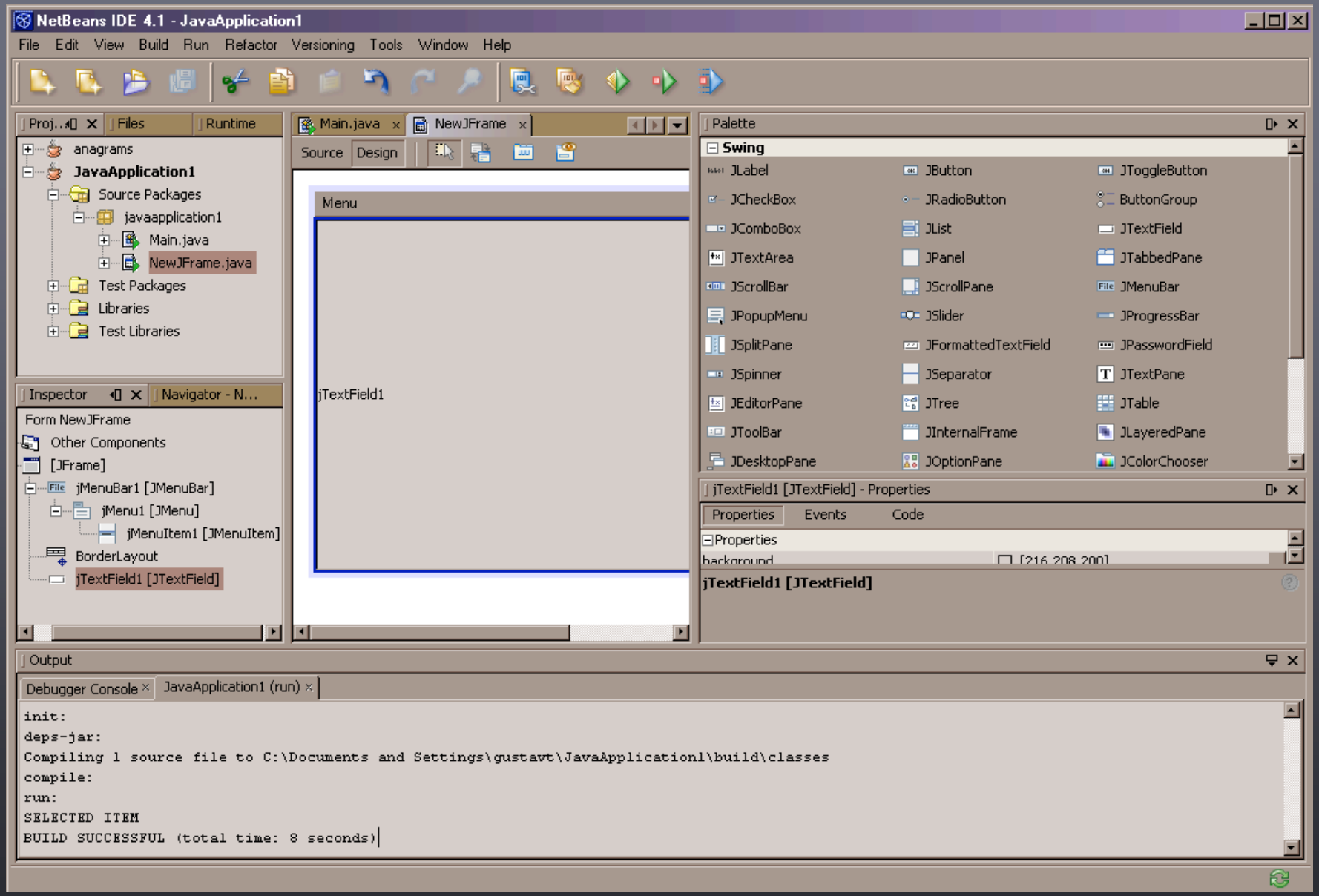
# GUI-byggare

- Ofta integrerade med en **IDE**, Integrated Development Environment
- GUI:er består oftast av widgets, samt kod som skickar meddelanden mellan dessa
- GUI-byggare: "rita" gränssnittet, kod som skapar widgets och deras layout genereras automatiskt!
- Sedan lägger man till händelselyssnare etc. själv, ofta via någon form av **property inspector**.

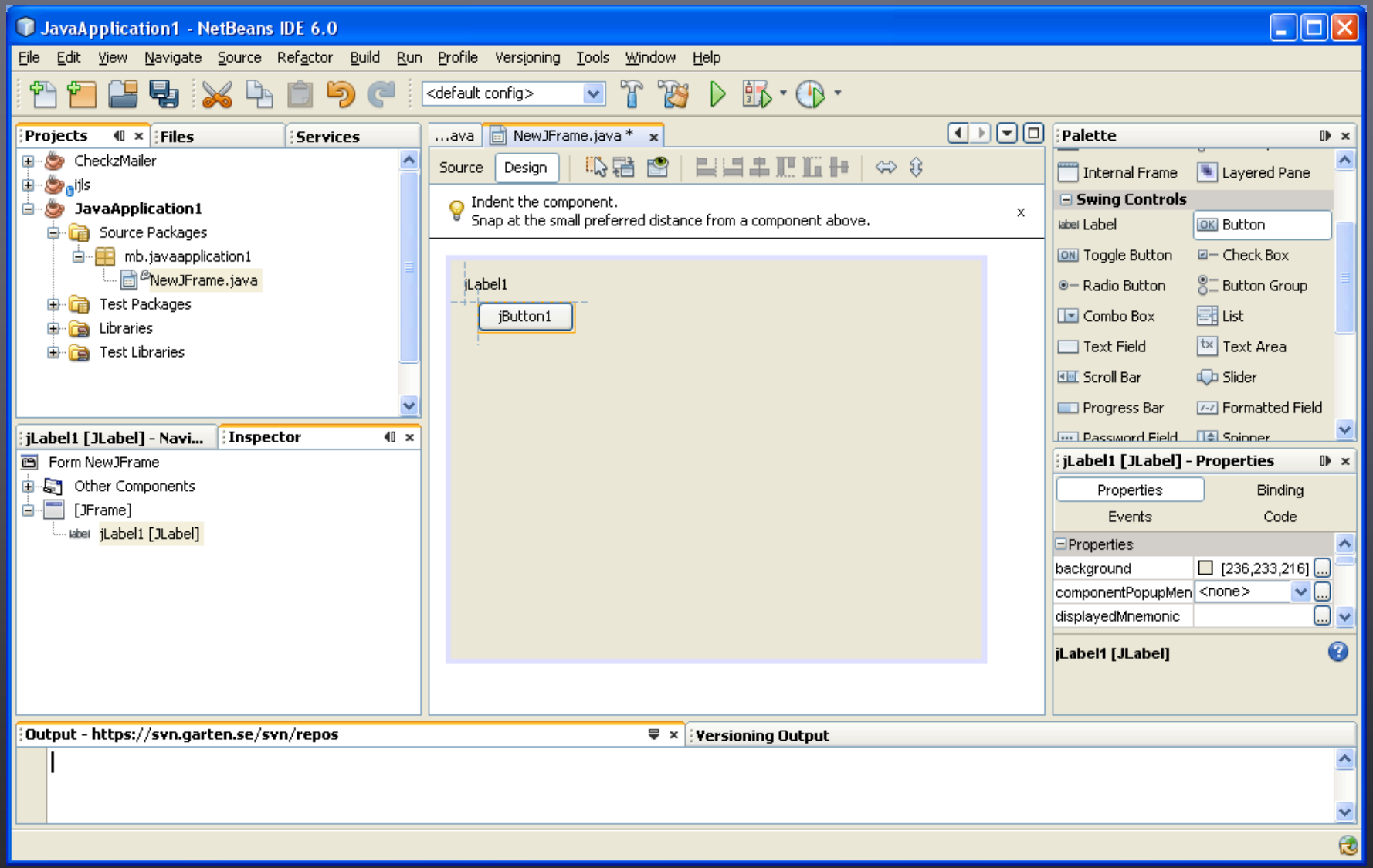
# Några GUI-byggare:

- **Visual Studio .NET Forms Designer**, www.microsoft.com
- **Borland C++ Builder**, www.borland.com
- **QT Designer**, www.trolltech.com
- **NetBeans**, www.netbeans.org
- Olika plug-ins för **Eclipse**, www.eclipse.org

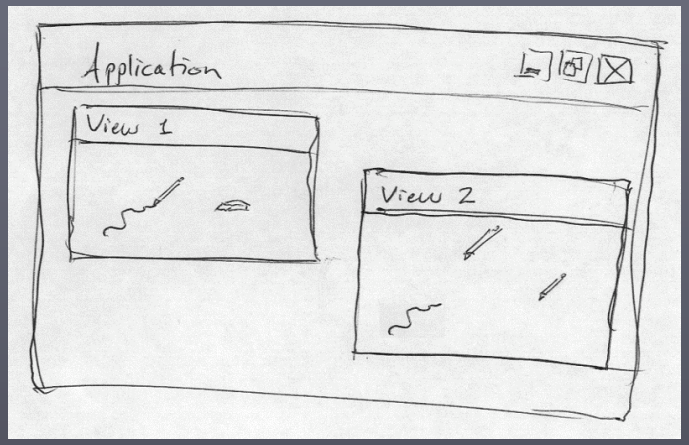
- Men det finns många, många fler!

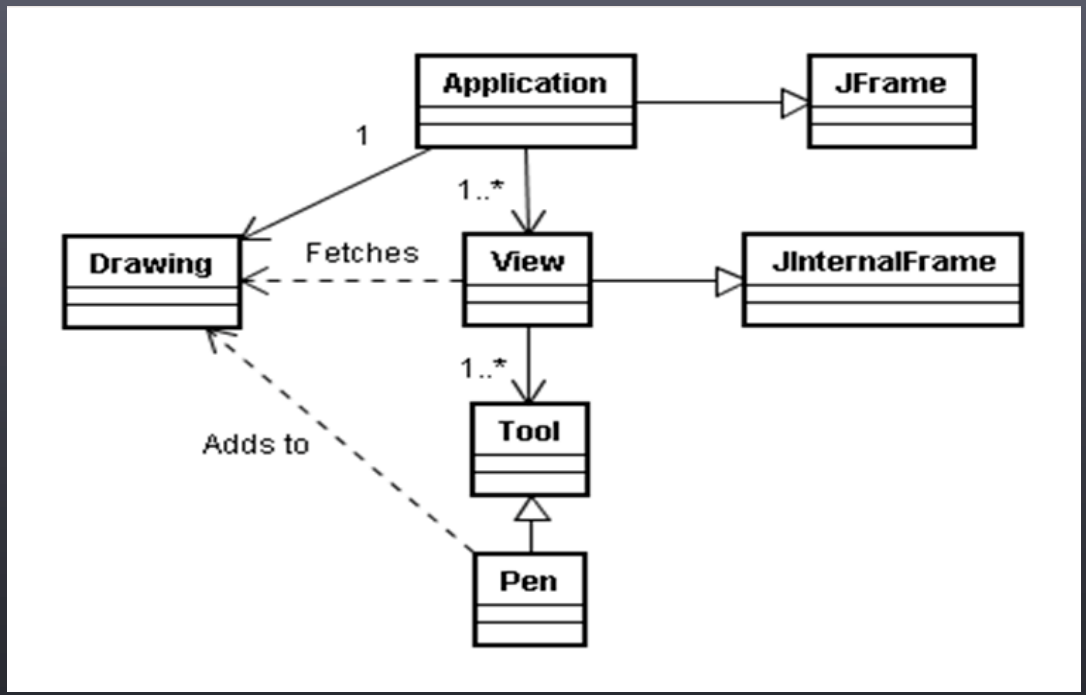

NetBeans IDE 4.1 - JavaApplication1
File Edit View Build Run Refactor Versioning Tools Window Help
Proj... Files Runtime
anagrams
JavaApplication1
Source Packages
javaapplication1
Main.java
NewJFrame.java
Test Packages
Libraries
Test Libraries
Inspector Navigator - N...
Form NewJFrame
Other Components
[JFrame]
jMenuBar1 [JMenuBar]
jMenu1 [JMenu]
jMenuItem1 [JMenuItem]
BorderLayout
jTextField1 [JTextField]
Main.java NewJFrame
Source Design
Menu
jTextField1
Palette
Swing
JLabel JButton JToggleButton
JCheckBox JRadioButton ButtonGroup
JComboBox JList JTextField
JTextArea JPanel JTabbedPane
JScrollBar JScrollPane JMenuBar
JPopupMenu JSlider JProgressBar
JSplitPane JFormattedTextField JPasswordField
JSpinner JSeparator JTextPane
JEditorPane JTree JTable
JToolBar JInternalFrame JLayeredPane
JDesktopPane JOptionPane JColorChooser
jTextField1 [JTextField] - Properties
Properties Events Code
Properties
jTextField1 [JTextField]
Output
Debugger Console JavaApplication1 (run)
init:
deps-jar:
Compiling 1 source file to C:\Documents and Settings\gustavt\JavaApplication1\build\classes
compile:
run:
SELECTED ITEM
BUILD SUCCESSFUL (total time: 8 seconds)

# Senaste NetBeans 6.0



# Java-labben


Application
View 1
View 2


Application
JFrame
1
1..*
Drawing
Fetches
View
JInternalFrame
1..*
Tool
Adds to
Pen

# Live-demo av NetBeans 6.0