GRIP 2000

Föreläsning 4

Innehåll

- Inmatningstyper
- Modeller för interaktiva applikationer
- Observer
- MVC
- Lyssnare
- Pluggbara komponenter
- Exempel

Olika inmatningstyper

- Request mode
  - Vänta tills dess att något sker
    - tex på inmatning via tangentbordet eller klick på mus
      - traditionell inmatning från tangentbord, scanf (C), readln (Pascal)

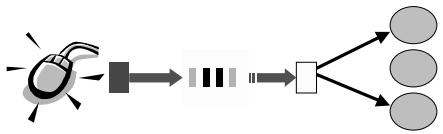
```
sensor.waitButton();
gc.displayString("Inmatning ", sensor.pos());
```

    - Där sensor är ett (tänkt) objekt som ger oss möjlighet att läsa av inmatningsverktygens status (dvs tex tangentbord och mus)
- Samplad/pollad
  - Kontrollera status för enhet
    - Omedelbar avläsning
    - tex är någon knapp på musen nertryckt

```
while(sensor.anyButtonPressed()) {
        pen.moveTo(sensor.mousePoint());
}
```


... inmatningstyper

- Händelsestyrd
  - Förändringar hos enheter placeras i en kö
    - intresserade program meddelas om att händelse inträffat från början av kön (flera kan alltså meddelas om samma händelse)



    - Det är sedan upp till varje program att själv bestämma hur det skall (eller inte skall) reagera
  - Exempel, en metod som reagerar om händelsen dubbeklick inträffar

```
public void doubleClickEvent(Event event) {
   frame.setLocation(event.position());
}
```


Modeller för konstruktion av interaktiva tillämpningar

- Modeller
  - Seeheim
  - PAC
  - MVC
- Tekniker
  - Call-back
  - Händelsebaserade
  - Adaptorer, listeners
  - Pluggbara
- ”Stil”
  - Direktmanipulativ, grafisk
  - Drag-and-drop
  - Objektorienterad

Modeller för konstruktion av interaktiva tillämpningar

- Separation av applikation och presentation
  - Newman's Reaction Handler, 1968
    - Var först med att separera applikation från presentation
- Seeheim, MVC och PAC
  - Seeheim, 1985
    - På en Workshop i Seeheim konstruerade man den första konceptuella arkitekturen för ett *User Interface Management System* (UIMS).
    - Man identifierade följande logiska komponenter i ett UIMS
      - presentation - visualisering, in- och utmatning
      - dialog kontroll - styr kommunikation mellan presentation och applikation
      - tillämpningsgränssnitt - "interfacesemantik" i applikation

…

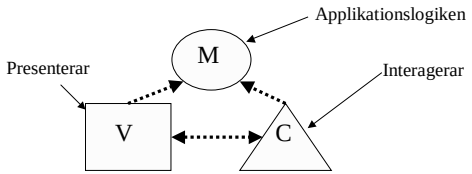
- Model View Controller (MVC), andra hälften av 1970-talet
  - Delar upp applikationerna i tre komponenter
    - applikationslogiken (modell)
    - presentation (vy)
    - interaktion (kontroll)
- Presentation Abstraction Control (PAC), 1989
  - Delar också upp applikationen i tre delar
    - abstraktion
    - presentation
    - kontroll
  - Mer konceptuell än MVC

Model View Controller (MVC)

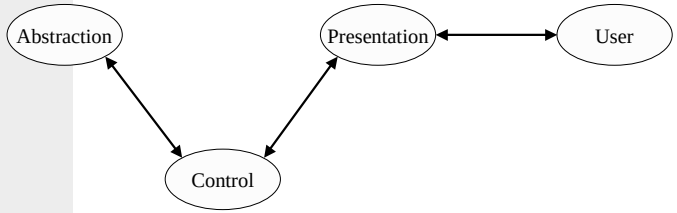
Model: logiken

View: presentationen

Controller: interaktionen



Presentation Abstraction Control (PAC)



, beroenden mellan objekt

Problem

Hur kan vi konstruera en mekanism som tillåter att vissa objekt meddelas om någon vital del förändras i andra objekt utan att objekten görs starkt knutna till varandra?

Krafter

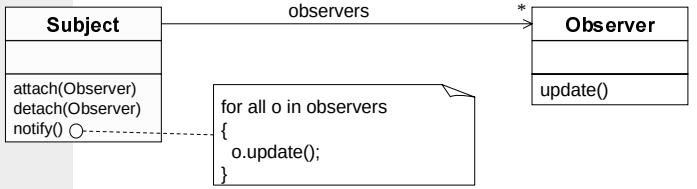
– Vi vill undvika stark koppling och beroende mellan objekten

– Intresserade objekt skall informeras om något förändras.

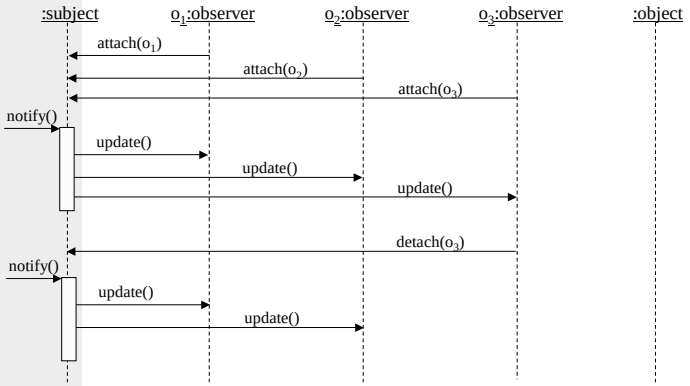
Lösning

Upprätthåll en lista av objekt som är beroende av ett visst objekt. Om objektet förändras skall dom beroende objekten meddelas. Dom beroende objekten skall vart och en själva avgöra hur dom skall reagera på förändringen.

...



…ett sekvensdiagram som visar ett typiskt scenario



…Javalösning: observer...

Javalösning med `Observer` och `Observable`.

I Java kan ett objekt som vill vara beroende av ett annat objekt implementera gränssnittet `Observer` medan det objekt som observeras görs till subklass till klassen `Observable` (eller möjligen använder ett fält som är subklass till denna typ).

- Gränssnittet `Observer`:

```
package java.util;
public interface Observer {
   void update(Observable o, Object arg);
}
```

… klassen …

```
package java.util;
public class Observable {
    private boolean changed = false;
    private Vector obs;

    private Observer[] arr = new Observer[2];

    public Observable() {obs = new Vector();}

    public synchronized void addObserver(Observer o) {
       if (!obs.contains(o)) {obs.addElement(o);}
    }

    public synchronized void deleteObserver(Observer o) {
       obs.removeElement(o);
    }
```

Metod för att Lägga till observer

Metod för att ta bort observer

… □□□□□□□□□□□forts…

```
public void notifyObservers() {notifyObservers(null);}

public void notifyObservers(Object arg) {
   int size=0;
   synchronized (this) {
      if (!hasChanged()) return;
      size = obs.size();
      if (size > arr.length) {arr = new Observer[size];}
      obs.copyInto(arr);
      clearChanged();
   }
   for (int i = size -1; i>=0; i--) {
      if (arr[i] != null) {
        arr[i].update(this, arg);
      }
   }
}
```

Metoder för att meddela att objektet ändrats

Om vi inte anger argument så läggs null på som argument

Meddelandet `update(…)` skickas till alla observers

Från vem

Argument till uppdateringen

… klassen forts …

```
public synchronized void deleteObservers() {
   obs.removeAllElements();
}

protected synchronized void setChanged()
   {changed = true;}

protected synchronized void clearChanged()
   {changed = false;}

public synchronized boolean hasChanged() {
       return changed;}

public synchronized int countObservers()
   {return obs.size();}
}
```

Sätt eller ta bort changed-flagga

… observer …

Exempel: MessageBoard och beroende studenter

```
import java.util.*;
class MessageBoard extends Observable {
  protected String message;

  public String getMessage(){return message;}

  public void changeMessage(String aMessage){
    message = aMessage;
    this.setChanged();
    this.notifyObservers(message);
  }
}
```

Subklassa Observable

Argumentet sparas (om tex något beroende objekt vill läsa av det)

Vi indikerar att objektet ändrats

Vi meddelar beroende objekt, med `message` som argument

… observer: exempel …

Vi antar att det redan finns en "vanlig" implementation av Student, som vi utökar

Implementera gränssnittet Observer

I `update(…)` definierar vi vad som skall göras då objektet får reda på att ett objekt som det är beroende av ändrats

```
import Persons.*;
import java.util.*;
class Student extends Persons.Student implements
  Observer {
  public void update(Observable o , Object arg){
        System.out.println(this.christianName() +
                           " tar emot meddelande: " + arg);
  }

  public Student(String christianName, String
  familyName, String pnr, String programme, String
  email) {
       super(christianName, familyName, pnr, programme,
  email);
  }
}
```


… observer: exempel …

```
public class TestObserver {
  public static void main(String [] args)
  {
   MessageBoard board = new MessageBoard();
   Student pers1 = new Student("Kalle", "Person", "133",
  "D96", "d96-kpe@nada.kth.se");
   pers1.addCourse("GRIP2001");
   pers1.addCourse("The Classic Malts");
   board.addObserver(pers1);
   board.changeMessage("Ny person: " +
                       pers1.christianName());
   /* Utskriften blir
   Kalle tar emot meddelande: Ny person: Kalle
   */
```

Gör pers1 beroende av `board`

Meddela att `board` ändrats

… observer: exempel …

```
   Student pers2 = new Student("Olle", "Olsson",
  "113", "D96", "olle@swipnet.se");
   pers2.addCourse("GRIP2001");
   pers2.addCourse("ENGLISH-1");
   board.addObserver(pers2);
   board.changeMessage("Ny person: " +
                       pers2.christianName());

   /* Utskriften blir
   Olle tar emot meddelande: Ny person: Olle
   Kalle tar emot meddelande: Ny person: Olle
   */
```

Gör pers2 beroende av `board`

Meddela att `board` ändrats

… observer: exempel

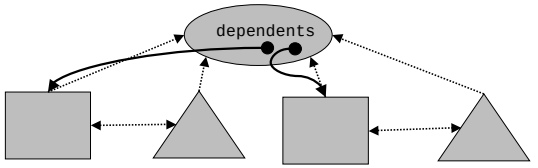
```
   Student pers3 = new Student("Lotta", "Andersson",
                               "123", "F97", "ff@home.se");
   pers3.addCourse("MDI01");
   pers3.addCourse("GRIP2001");
   board.addObserver(pers3);
   board.changeMessage("Ny person: " +
                       pers3.christianName());

   /* Utskriften blir
   Lotta tar emot meddelande: Ny person: Lotta
   Olle tar emot meddelande: Ny person: Lotta
   Kalle tar emot meddelande: Ny person: Lotta
   */
  }
}
```

Model View Controller (MVC)

- Baseras på mönstret Observer
  - Vyerna observerar (prenumererar på) förändringar hos modellen (publicisten)
  - Annorlunda uttryckt:
    - En vy görs beroende (eng. dependent) av modellen
    - Vid intressanta förändringar i modellen så meddelas dom beroende objekten (dvs vyerna)
  - Ren struktur
    - Modellen skall inte ha någon direkt vetskap om vilka eller hur många vyer som är beroende av den
    - Modellen skall inte heller ta hänsyn till hur dess data presenteras eller hur man interagerar med den

MVC (Konkret struktur, med principkod)

- Modellen har *implicita referenser* till vyerna



- vyerna blir automatisk beronde då modellen anges via meddelandet `model(Model aModel)`

```
view.model(aModel)
```


MVC (Modellen ändras) - principiell kod

- Vid förändring görs (alternativt)

```
model.changed()
model.changed(argument)
model.changed(argument, parameter)
```

- Vyn får meddelandet

```
update(argument, parameter, avsändare)
```

- Denna metod finns i klassen objekt så om man inte vill ta hand om argument räcker det att implementera

```
update(argument, parameter)
```

- eller

```
update(argument)
```


Sammanfattning konstruktion med View-Controller (typfall)

1 Skapa klasser
- Skapa applikationslogiken
  - Oftast bäst att fokusera på och konstruera den logiska applikationen, utan interaktion och presentation, först
  - Frågor man kan ställa sig: Vad är modell? Vad är ”bara” hjälpklasser?
- Skapa vy
  - (I Java som direkt eller indirekt sublass till `JComponent` i Smalltalk som direkt eller indirekt subklass till `View`)
- Skapa kontrollklass
  - I Java används en kombination av lyssnare och egna klasser i Smalltalk (som vi ska se senare i kursen) finns speciella kontrollklasser
- Gör kontrollern till defaultklass för vyn
  - Detta gäller främst för den puristiska implementationen av MVC i Smalltalk, i Java får vi som sagt bla utnyttja lyssnare
- Sätt ihop det hela mha Observermönstret
  - I Java måste man själv se till att koppla ihop det hela. I Smalltalk ser vyn till att rätta beroenden sätts upp då man ger den en modell

…

2 Definiera hur utritning går till

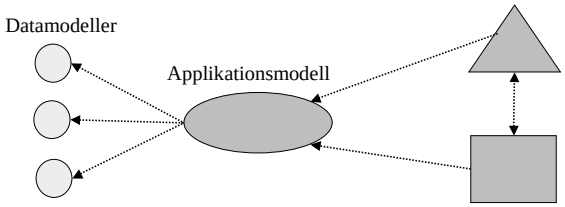
- steg 1 skriv metoden Java: `paint(Graphics)` eller `paintComponent(Graphics)` i Smalltalk `displayOn`: i vyn
- senare då vi använder kan vi vid behov göra lite mer sofistikerade saker

3 Definiera hur inmatning från interkationsverktygen

- Java: konstruera händelselyssnare och lämpliga metoder
- Smalltalk: skriva metoder med givna namn som tar emot händelser av olika typer

Problem med MVC

- Ibland svårt att definitivt separera funktionalitet mellan modell, vy och kontroller
- Ibland kan det vara bra att dela upp applikationen i *datamodeller* och *applikationsmodell*



Applikationsmodellen kan då innehålla kod som är relaterad till användargränssnittet, dvs controller och view
men fortfarande bevaras den lösa kopplingen till datamodellen

MVC i JAVA

`class MinModell extends Observable`

- lägg till UI-objekt som prenumerant, dvs `addObserver(Observer o)`
- skicka `changed`, dvs

```
setChanged();
notifyObservers();
```

till modellobjektet om intressant ändring skett

`class MinUIClass extends Component implements Observer`

- implementera `update(Obsevervable o, Object x)`
  - uppdatera skärmen

Java Eventmodell 1.1

- Java 1.1 och senare har en modell där man definierar objekt som tar omhand om händelser som sker i komponenter i gränssnitten
- I princip sätter man upp en händelselyssnare per typ (tangentbord, mus, fokus, etc.) av händelse man är intresserad av i appleten
- Varje komponent (knapp, fält, mfl) kan också hanteras av sin egen händelselyssnare

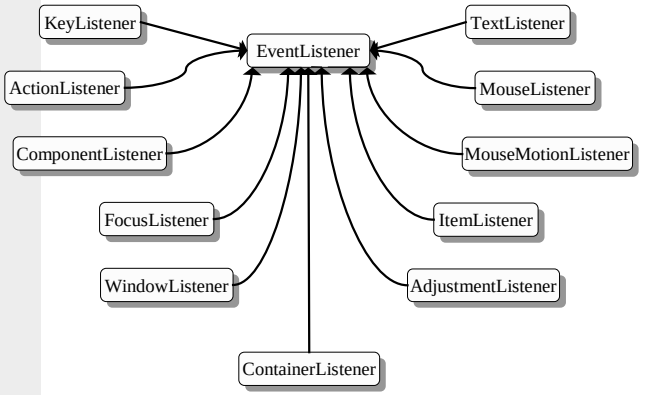
```
buttonX.addActionListener(myListenerForButtonX)
```

- Med fördel används anonyma klasser för att definiera lyssnare för olika objekt

Händelselyssnare

- Händelser till ett fönster eller komponent tas omhand av händelselyssnare
  - påminner mycket om mönstret `Observer`, och i grafiska sammanhang om MVC, fast dom "beroende" objekten, dvs lyssnarna, måste implementera ett mer varierat utbud av metoder
    - beroende av typ av lyssnare måste olika API:er implementeras
- Olika typer av lyssnare tar hand om olika typer av händelser, `MouseListener`, `WindowListener`, `FocusListener`, `ActionListener`, osv
  - Namnet är uppbyggt på följande sätt: **Händelsetyp***Listener*
- Till varje lyssnare finns ett speciellt interface som beskriver vilka metoder som lyssnaren implementerar
- Lyssnarna och deras interface tillhör ett gemensamt package nämligen: `java.awt.event`

Händelselyssnare, några typiska



Exempel: implementation av lyssnare direkt

```
package GRIP2001F4;
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.setTitle("Fönster som lyssnar efter fönsterhändelser v1 (fönstret gör det
    själv)");
        frame.setSize(400, 300);
        frame.addWindowListener(frame);
        frame.setVisible(true);
    }
}
```

Exempel: lyssnare som inre klass

```
package GRIP2001F4;
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) {}
}
```

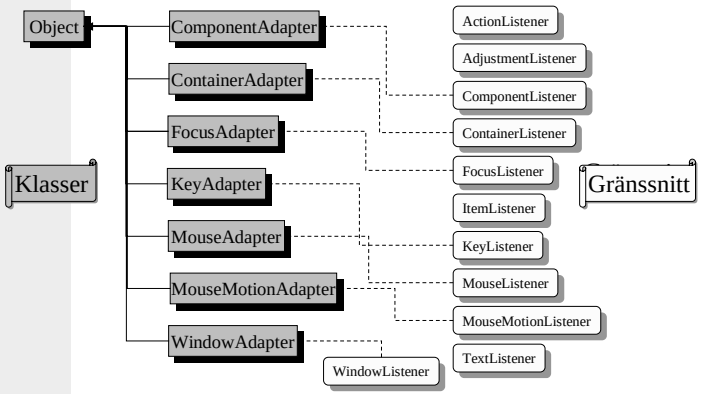


```
...
public class MyFrame2 extends Frame {
    public static void main(String [] args) {
        Frame frame = new MyFrame2();
        frame.setTitle("Fönster som lyssnar efter fönsterhändelser v2 (med inre klass)");
        frame.setSize(400, 300);
        frame.addWindowListener(new MyWindowListener());
        frame.setVisible(true);
    }
}
```


Adaptorer och händelselyssnare

- För att förenkla användningen av händelselyssnare finns ett antal fördefinierade adaptorer
  - Dessa adaptorer *implementerar alla metoder* deklarerade i den aktuella lyssnarens gränssnitt
  - En adaptor implemeneterar *endast tomma metoder*
  - Genom att *subklassa en adaptor* och använda den som lyssnare så *räcker det att endast skriva om dom metoder som verkligen gör något,* så slipper man som i föregående exempel implementera massa metoder utan metodkropp
  - Dessa adaptorer är speciellt enkla att använda tillsammans med *anonyma inre klasser*

Adaptorer



Exempel: lyssnare via adapter

```
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.setTitle("Fönster som lyssnar efter fönsterhändelser v3 (med egen
    adapter)");
        frame.setSize(400, 300);
        frame.addWindowListener(new MyWindowAdapter());
        frame.setVisible(true);
    }
}
```


Exempel: lyssnare via anonym subklass

```
public class MyFrame4 extends Frame {
    public static void main(String [] args) {
        Frame frame = new MyFrame4();
        frame.setTitle("Fönster som lyssnar efter fönsterhändelser v4 (med
    anonym subklass till adapter)");
        frame.setSize(400, 300);
        frame.addWindowListener(new WindowAdapter (){
                public void windowClosing(WindowEvent e) {
                        System.exit(0);
                }
        });
        frame.setVisible(true);
    }
}
```


Kommunicera och prenumerera på förändringar (en kort repetition)

- Traditionellt
  - callbacks
    - dvs en funktion kopplas till att ta emot en viss händelse från systemet eller komponent
    - OpenGL, X
    - ☺ enkelt
    - ☹ statiskt
  - MVC
    - baseras på Observer/beroenden
    - alla typer av förändringar via samma ”kanal”
      - möjligen parameterstyrt
    - ☺ generellt
    - ☺ (språk-) neutralt
    - ☺ skalbart/distribuerbart
    - ☹ en ”kanal” med dispatching hos klient

•••

- Lyssnare och adaptorer
  - ett helt objekt kopplas till en viss typ av händelse
    - ☺ högre nivå än call-backs
    - ☹ lite komplext (och fördolt) beteende
      - ☺ mest ett problem för ovana användare av det aktuella systemet
  - finns stöd för detta i
    - Java
      - med fördefinierade listener-interface och adaptorer
        - specialiserade
      - kopplade främst till grafiska komponenter
    - Smalltalk
      - mer generell än Javas modell
        - alla objekt förberedda
        - går att pluggas till en speciell del (tex en instansvariabel)
        - men också mer komplex (och fördold för den oinvigde)
      - MVC i botten

Lyssnare och adaptorer: implementation

- I Java är många komponenter, som vi tidigare nämnt, förberedda för att hanteras av speciella lyssnare
  - I princip går det till på följande sätt
    - en komponent registrerar ett visst objekt som en lyssnare på vissa typer av händelser
    - när en händelse av denna typ inträffar så vidarbefordras den i form av ett förutbestämt (deklarerat i ett Interface) meddelande till lyssnaren
    - lyssnaren i sin tur är vanligen en applikation eller en adaptor som i sin tur kommunicerar med applikationen

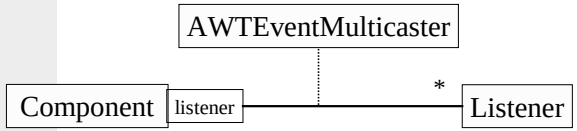
registrera lyssnare

händelser tas omhand av lyssnare

lyssnaren beskriver beteende och kommunicerar med applikationen

...

- Centralt är klasserna
  - Component med subklasser som Button och TextField
    - Definierar metoder som addActionListener
      - dvs binder en lyssnare
  - AWTEventMulticaster
    - Hanterar bindning mellan applikation och komponent
    - Skickar förutbestämt meddelande till lyssnare vid given händelse



Exempel: definiera egen lyssnarklass

```
import java.util.EventListener;
import java.awt.event.ActionEvent;

public interface MyActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```


...(MyComponent)

```
package GRIP2001F4;
import java.util.Vector;
import java.util.Enumeration;
import java.awt.event.ActionEvent;

public class MyComponent {
    Vector actionListeners = new Vector();

    public void addMyActionListener(MyActionListener listener) {
        if(!actionListeners.contains(listener)) actionListeners.add(listener);
    }

    public void removeMyActionListener(MyActionListener listener) {
        if(actionListeners.contains(listener)) actionListeners.remove(listener);
    }
```

```
...
    public void fireMyAction() {
        ActionEvent event = new ActionEvent(this, 4711, "My action fired!");
        Vector cloneOfListeners = (Vector) actionListeners.clone();
        for(Enumeration e = cloneOfListeners.elements(); e.hasMoreElements();)
            ((MyActionListener) e.nextElement()).actionPerformed(event);
    }
}
```


... (MyListenerTest)

```
package GRIP2000F4;
import java.awt.event.ActionEvent;
class MyLocalActionListener implements MyActionListener {
    public void actionPerformed(ActionEvent e) {
      System.out.println("ActionEvent received: " + e);
    }
}
public class MyListenerTest {
    public static void main(String [] args) {
      MyComponent component = new MyComponent();
      component.addMyActionListener(new MyLocalActionListener());
      component.addMyActionListener(new MyLocalActionListener());
      component.addMyActionListener(new MyActionListener() {
        public void actionPerformed(ActionEvent e) {
          System.out.println("ActionEvent received in anonymous class: " + e);
        }});
      component.fireMyAction();
    }
}
```


Exempel: RitProgram1

```
package GRIP2001F4;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class RitProgram1 extends JFrame
    implements WindowListener, MouseListener,  MouseMotionListener {
        private int last_x, last_y;

        public RitProgram1(String s) {
            super(s);
            this.init();
        }
```



```
...
        public void init() {
            // Använd MouseListener och MouseMotionListener
            // addXXXListener anger att vi ar intresserade av den aktuella typen av
    händelser.
            this.addWindowListener(this);
            this.addMouseListener(this);
            this.addMouseMotionListener(this);
            this.setSize(300, 300);
        }

        // En metod beskriven i MouseListener.
        // Anropas då vid tryck på musknapp.
        public void mousePressed(MouseEvent e) {
            last_x = e.getX();
            last_y = e.getY();
        }
```

```
...

        // Metoder deklarerade i MouseListener måste implementeras
        // även om vi inte gör något i dem
        public void mouseReleased(MouseEvent e) {}
        public void mouseClicked(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}

        // och samma sak för metoder från MouseMotionListener.
        public void mouseMoved(MouseEvent e) {}

        // En metod beskriven i MouseMotionListener.
        // Anropas då musen rörs med knapp nertryckt.
        public void mouseDragged(MouseEvent e) {
                Graphics g = this.getGraphics();
                int x = e.getX(), y = e.getY();
                g.drawLine(last_x, last_y, x, y);
                last_x = x; last_y = y;
        }
```



```
...
        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 []) {
                RitProgram1 ritprog = new RitProgram1("Enkelt ritprogram");
                ritprog.setVisible(true);
        }
```


Exempel: RitProgram2

```
package GRIP2001F4;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class MinRityta extends JComponent {
    protected int last_x, last_y;
    public MinRityta() {
        super();
        this.init();
    }
```



```
...
    public void init() {
        this.addMouseListener(new MouseAdapter () {
            public void mousePressed(MouseEvent e) {
                last_x = e.getX();
                last_y = e.getY();
            }
        });
        this.addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                Graphics g = getGraphics();
                int x = e.getX(), y = e.getY();
                g.drawLine(last_x, last_y, x, y);
                last_x = x; last_y = y;
            }
        });
    }
}
```

```
...
public class RitProgram2 {
    public static void main(String args []) {
        JFrame ritprog = new JFrame("Ritprogram 2");
        ritprog.getContentPane().add(new MinRityta());

        ritprog.addWindowListener(new WindowAdapter () {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        ritprog.setSize(300, 300);
        ritprog.setVisible(true);
    }
}
```


Exempel: RitProgram3 (dubbelbuffring)

```
package GRIP2001F4;
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class MinRityta extends JComponent {
    Image image = null;
    Graphics imG = null;

    protected int last_x, last_y;
    public MinRityta() {
        super();
        this.init();
    }
```



```
        ...
    protected Graphics getImGraphics() {
        if(image == null) {
            Dimension dim = getSize();
            image = createImage(dim.width, dim.height);
            imG = image.getGraphics();
        }
        return imG;
    }

    public void paint(Graphics g) {
        if(!(image == null)) g.drawImage(image, 0, 0, this);
    }
```



```
...
    public void init() {
        this.addMouseListener(new MouseAdapter () {
            public void mousePressed(MouseEvent e) {
                last_x = e.getX();
                last_y = e.getY();
            }
        });
        this.addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent e) {
                Graphics g = getImGraphics();
                int x = e.getX(), y = e.getY();
                g.drawLine(last_x, last_y, x, y);
                repaint();
                last_x = x; last_y = y;
            }
        });
    }
}
```

```
...
public class RitProgram3 {
    public static void main(String args []) {
        JFrame ritprog = new JFrame("Ritprogram 3");
        ritprog.getContentPane().add(new MinRityta());

        ritprog.addWindowListener(new WindowAdapter () {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        ritprog.setSize(300, 300);
        ritprog.setVisible(true);
    }
}
```


Fler exempel i appendix

- RitProgram4 med effektivare uppdatering
- RitProgram5 med flera fönster baserat på MVC
- RitProgram6 som 5 fast alternativt sätt att reagera på stängningshändelser till fönstret
- RitProgram7 som 6 fast lite olika vyer

- Dessa exempel tittar vi också vidare på/repeterar under övning 2 och 3