Labbinstruktioner för Java/Swing

Grafik- och interaktionsprogrammering 2008
Martin Berglund <mabe02@kth.se>

Allmänt

Dessa instruktioner är på intet sett den enda möjliga lösningen på labben, tvärtom finns det många olika lösningar. Att följa instruktionerna är helt frivilligt och de representerar ett möjligt sätt att angripa problemet med en MVC-lösning. Frågor angående dessa instruktioner besvaras under bokad laborationstid. Författaren hoppas att de ska vara både enkla att förstå och lätta att följa.

1. Modellen

Börja med att skapa en klass för vår bakomliggande modell. Vi döper klassen till *MyModel*. Låt klassen ha en privat medlemsvariabel av typen *BufferedImage*, vi döper variabeln till *myBackend*. Initiera *myBackend* i konstruktorn till *MyModel* och sätt storleken till ett lämpligt värde, exempelvis 500 pixlar i höjd, 500 pixlar i bredd. Som typ, använd *BufferedImage.TYPE_INT_ARGB*.

Nu fyller vi på med lite metoder som gör att andra klasser också kan interagera med vår model. Skapa en metod *getWidth*() och en *getHeight*() som returnerar bredden respektive höjden på vår modell (du kan returnera samma tal som du satte högre upp, men visst är det snyggare att returnera *myBackend.getWidth*() och *myBackend.getHeight*() istället?). Därefter behöver vi ett sätt att rita ut vår modell på en yta, vilken som helst. Här är ett förslag:

```
public void drawModelOnSurface(Graphics g)
{
        g.drawImage(myBackend, 0, 0, null);
}
```

Idén är att andra klasser skickar in sina *Graphics*-objekt till vår modell och modellen ritar ut sig själv på den.

Slutligen behövs det något sätt att utifrån manipulera modellen. Eftersom det är vår *myBackend* som är vår data så kan vi lämna ut ett *Graphics*-objekt för att manipulera den. Så här skulle det kunna se ut:

```
public Graphics2D getModelModifier()
{
        return (Graphics2D)myBackend.getGraphics();
}
```

Innan vi går vidare från modellen kan det vara bra att se till att den är helt vit till att börja med. Lägg till följande kod efter initieringen i konstruktorn så borde det fixa sig:

```
Graphics g = myBackend.getGraphics();
g.setColor(Color.WHITE);
g.fillRect(0, 0, myBackend.getWidth(), myBackend.getHeight());
```

2. Huvudfönstret

Skapa en ny klass för huvudfönstret, kalla den *MyFrame*. Låt den nya klassen ärva av *JFrame*. Skapa en konstruktor till klassen och som första rad anropa

```
super(”GRIP-lab”);
```

eller vilken text som du nu vill ha som titel på det nya fönstret. Lägg till *main*-metoden här, den skulle kunna se ut så här:

```
public static void main(String []args)
{
        new MyFrame().setVisible(true);
}
```

Om programmet nu körs så kommer vårt nya fönster att visas. Lägg därefter till en medlemsvariabel till klassen av typen *JDesktopPane* med namn *myDesktop*. En *JDesktopPane* är en GUI-komponent som är till för att lägga interna fönster på. Instansiera den i konstruktorn. Eftersom den ska fylla upp hela vårt huvudfönster slipper vi *LayoutManager*s och kan göra så här istället:

```
setContentPane(myDesktop);
```

Så kan vi sätta storleken på huvudfönstret med metoden *setSize(int width, int height)* till något lämpligt, exempelvis 640 x 480.

3. Huvudmenyn

Vi fyller på med en huvudmeny i konstruktorkoden till huvudfönstret och skapar lite menyalternativ, exempelvis så här:

```
JMenuBar menuBar = new JMenuBar();
JMenu fileMenu = new JMenu("File");
JMenu sizeMenu = new JMenu("Size");
JMenu colorMenu = new JMenu("Color");
JMenu newToolMenu = new JMenu("New tool");

JMenuItem newWindowItem = new JMenuItem("New window");
JMenuItem size1Item = new JMenuItem("1");
JMenuItem size5Item = new JMenuItem("5");
JMenuItem size10Item = new JMenuItem("10");
JMenuItem colorRedItem = new JMenuItem("Red");
JMenuItem colorBlueItem = new JMenuItem("Blue");
JMenuItem colorBlackItem = new JMenuItem("Black");
JMenuItem colorGreenItem = new JMenuItem("Green");

menuBar.add(fileMenu);
menuBar.add(sizeMenu);
menuBar.add(colorMenu);
menuBar.add(newToolMenu);

fileMenu.add(newWindowItem);
sizeMenu.add(size1Item);
sizeMenu.add(size5Item);
sizeMenu.add(size10Item);
colorMenu.add(colorRedItem);
colorMenu.add(colorBlueItem);
colorMenu.add(colorBlackItem);
colorMenu.add(colorGreenItem);

setJMenuBar(menuBar);
```

Då vill vi ha någonting som kan ta emot händelser från när användaren klickar på någon av dessa. Den som tar emot händelserna måste vara av typen *ActionListener*, och vi kan låta vår huvudklass sköta detta. Låt *MyFrame* implementera *ActionListener* och skapa metoden

```
public void actionPerformed(ActionEvent e)
```

som implementeringen kräver. Länka sedan alla menyalternativ i konstruktorn till vår nydefinierade händelsehanterare. Exempelvis så här:

```
newWindowItem.addActionListener(this);
size1Item.addActionListener(this);
size5Item.addActionListener(this);
size10Item.addActionListener(this);
colorRedItem.addActionListener(this);
colorBlueItem.addActionListener(this);
colorBlackItem.addActionListener(this);
colorGreenItem.addActionListener(this);
```

Vi måste kunna skilja på händelserna också, så ge de olika knapparna olika *ActionCommand*. Exempelvis:

```
newWindowItem.setActionCommand("new window");
size1Item.setActionCommand("size");
size5Item.setActionCommand("size");
size10Item.setActionCommand("size");
colorRedItem.setActionCommand("color");
colorBlueItem.setActionCommand("color");
colorBlackItem.setActionCommand("color");
colorGreenItem.setActionCommand("color");
```

Sätt slutligen en print-sats i *actionPerformed* som skriver ut vilket *actionCommand* som användes och testa så att programmet fungerar!

4. Vyer

Skapa en ny klass för vårt interna fönster, döp den till *MyInternalFrame*. Ärv av *JInternalFrame* och skapa en konstruktor. Sätt första raden till

```
super("Vy", false, true, false, true);
```

så får vi en snygg titel på våra interna fönster. För de övriga parametrarna hänvisar jag till officiell dokumentation.

Större delen ska täckas av rit-ytan, men den gör vi bäst som en egen komponent, så skapa ännu en ny klass som vi döper till *MyView*. Låt den ärva av *JComponent*. Skapa en tom konstruktor. Vi ska använda vår modell som källdata för vyn, men låt oss först testa om komponenten fungerar. Vi överlagrar *paint*(…) eller *paintComponent*(…) för att kunna styra över dess utseende när komponenten ska ritas ut på skärmen. I vår labb är det ingen större skillnad på vilken av de två vi överlagrar, men Sun anser att vi ska använda *paintComponent* så låt oss göra det. Här är ett exempel som ritar lite skräp på komponentens yta:

```
protected void paintComponent(Graphics g)
{
        Random random = new Random();
        for(int y = 0; y < getHeight(); y++)
        {
```

```
                    for(int x = 0; x < getWidth(); x++)
                    {
                              g.setColor(new Color(random.nextInt(255),
                                                  random.nextInt(255),
                                                  random.nextInt(255)))
                                                  ;
                              g.drawRect(x, y, 1, 1);
                    }
          }
}
```

Vi vill nu att våra interna fönster ska ha en *MyView*-komponent som täcker hela deras yta. Vi får göra det med en *BorderLayout*. Lägg till det här till *MyView*-komponentens konstruktor:

```
getContentPane().setLayout(new BorderLayout());
getContentPane().add(new MyView(), BorderLayout.CENTER);
pack();
```

Innan vi kan testa måste vi kunna lägga till nya interna fönster till vårt huvudfönster. Vi återgår till *actionPerformed*(…) i *MyFrame* och lägga till just den funktionaliteten:

```
if(e.getActionCommand().equals("new window"))
{
          MyInternalFrame internalFrame =
                                  new MyInternalFrame();
          internalFrame.setVisible(true);
          myDesktop.add(internalFrame);
}
```

Nu borde programmet vara klart att testas!
Om du tycker att de interna fönstrena är för små kan du lägga till följande rad i *MyView*-konstruktorn:

```
setPreferredSize(new Dimension(200, 200));
```

men se till att ta bort den sen!

5. Koppla vyn till modellen

Istället för skräp vill vi att *MyView* ska visa vad som finns i modellen. Dessutom vill vi att modellen ska bestämma storleken på *MyView*, så att de matchar. Börja med att skapa en medlemsvariabel till *MyFrame* som är privat, av typen *MyModel* och som får namnet *myModel*. Instansiera den i konstruktorn. Vi vill kunna passa ner den till varje vy-komponent, så modifiera konstruktorn till *MyInternalFrame* så att den ta emot ett argument av typ *MyModel*. Ändra så koden som skapar nya *MyInternalFrame*-objekt så den skickar med vår medlemsvariabel:

```
MyInternalFrame internalFrame = new MyInternalFrame(myModel);
```

Därefter i konstruktorn till *MyInternalFrame*, skicka vidare den till vår *MyView*:

```
getContentPane().add(new MyView(myModel), BorderLayout.CENTER);
```

Modifiera konstruktorn I *MyView* så den tar emot en *MyModel* och spara undan den i en egen medlemsvariabel, exempelvis såhär:

```
public class MyView extends JComponent
...
          private MyModel myModel;
          ...
          public MyView(MyModel myModel)
          {
```

```
this.myModel = myModel;
...
```

Nu ska vi ändra på *paintComponent*-metoden så att den istället ritar ut vår modell. I steg 1 skapade vi en metod i modell-klassen som gör att den ritar ut sig på en annan yta. Låt oss använda den!

```
protected void paintComponent(Graphics g)
{
        //Töm komponenten:
        g.setColor(Color.WHITE);
        g.fillRect(0, 0, getWidth(), getHeight());

        //Skicka vidare till modellen
        myModel.drawModelOnSurface(g);
}
```

Så borde vi sätta komponentens storlek till samma som modellen. Vi lägger till i konstruktorn:

```
setSize(myModel.getWidth(), myModel.getHeight());
```

Testa igen! De interna fönstrena borde nu bli så stora som du satte modellen och vara helt vita, eftersom modellen är helt vit. Låt oss ändra på den saken!

6. Verktyg

Låt oss börja med en gemensam basklass för verktyg som kan manipulera bilden. Vi döper den till *MyTool*. Låt den vara abstract med en konstruktor som är protected. Låt den ha följande medlemsvariabler:

```
private MyModel myModel;
private int x;
private int y;
private int size;
private Color color;
```

och låt konstruktorn ta emot och sätta dessa värden. Skapa sedan get-metoder och set-metoder för dessa variabler, undantaget *myModel* som bara behöver en get-metod, som kan vara *protected* (så ingen annan klass pillar på den). Skapa därefter en abstrakt metod *getIcon*() som returnerar en Image, det får vara ikonen för verktyget. Låt metoden vara *protected* så kan ingen utomstående klass komma åt ikonen. Vi kan därefter skriva en metod som ritar ut vårt verktyg på en yta:

```
public void paintToolOnSurface(Graphics g)
{
        g.drawImage(getIcon(), x, y, null);
}
```

För att kunna interagera med användaren måste vårt verktyg kunna ta emot händelser såsom när vi klickar och flyttar musen. Låt oss därför lägga till några fler abstrakta metoder:

```
public abstract void mouseDown();
public abstract void mouseUp();
public abstract void mouseDragged();
```

Vi vill att varje vy ska ha sin egen uppsättning verktyg på skärmen så lägg till en medlemsvariabel till klassen *MyView* av typen *ArrayList<MyTool>*, vi kan kalla den tools. Om det inte fungerar att definiera en *ArrayList* med generics (’<’ och ’>’) så beror det på att du kör en gammal version av Java (< 1.5) och du rekommenderas uppgradera eller anpassa

koden därefter. Instansiera variabeln i konstruktorn. Skapa en metod för att lägga till verktyg till vyn, exempelvis

```
public void addTool(MyTool newTool)
```

som lägger till verktyget i listan. Så ska vi rita ut alla verktyg när vi ritar komponenten. Eftersom vi vill att verktygen ska ligga överst gör vi det sist i *paintComponent*-metoden, såhär kanske:

```
...
for(MyTool tool: tools)
          tool.paintToolOnSurface(g);
...
```

Nu kan vi alltså lägga ut abstrakta verktyg på vår komponent men vi kanske fortfarande inte interagera med dem.

7. Händelser från användaren

Låt oss börja med att fånga händelser från musen som rör vår vykomponent. Låt därför *MyView* implementera *MouseListener* och *MouseMotionListener*. Här skulle vi kunna låta en intern klass sköta indatahanteringen, det skulle bli mer MVC, men i vårt exempel kan vi låta *MyView* ta hand om det. Implementera alla metoder som är obligatoriska i och med de två interfacen men lämna de tomma tills vidare. Gör kopplingen mellan komponenten och sig själv som hanterare genom att lägga till i konstruktorn:

```
addMouseListener(this);
addMouseMotionListener(this);
```

Användaren kan bara använda ett verktyg åt gången per vy så vi kommer att behöva en variabel som håller reda på vilken (om någon) som är den aktiva. Deklarera denna variabel som *activeTool* av typen *MyTool* och gör den privat i klassen. Sätt den till *null* i konstruktorn. Låt oss då bestämma hur verktygen ska styras.

- Vänstra musknappen väljer verktyg (om inget verktyg är aktivt) eller ritar (om ett verktyg är aktivt)
- Högra musknappen släpper ett verktyg (om ett verktyg är aktivt) eller gör ingenting (annars)

Så, vi måste kunna veta när användaren har klickat på ett verktyg. Låt oss bestämma oss för att alla verktyg har storlek 32x32 pixlar och skriva följande metod till *MyView*:

```
private MyTool pickupTool(int mouseAtX, int mouseAtY)
{
          for(MyTool tool: tools)
          {
                    if(mouseAtX >= tool.getX() &&
                              mouseAtX < tool.getX() + 32 &&
                              mouseAtY >= tool.getY() &&
                              mouseAtY < tool.getY() + 32)
                    {
                              return tool;
                    }
          }
          return null;
}
```

Så kan vi skriva till vår mouseClicked-metod:

```
if(e.getButton() == MouseEvent.BUTTON1 &&
                        activeTool == null)
{
          activeTool = pickupTool(e.getX(), e.getY());
          if(activeTool != null)          //Fick vi något?
          {
                    //Det här gömmer muspekaren!
                    setCursor(getToolkit().createCustomCursor(
                              new BufferedImage(3, 3,
                              BufferedImage.TYPE_INT_ARGB),
                                        new Point(0, 0),
                                         "null"));
          }
}
else if(e.getButton() == MouseEvent.BUTTON3 &&
                        activeTool != null)
{
          activeTool = null;
          setCursor(Cursor.getDefaultCursor());
}
```

Vi kollar alltså vilken musknapp som har klickats och plockar upp respektive släpper verktyg. Då ska vi bara hantera förflyttningar och ritoperationer så är vi nästan färdiga!

Om musen förflyttas över vyn när vi har ett verktyg valt så ska verktyget förflyttas med musen. Metoden *mouseMoved* anropas när musen rör sig över komponenten utan att någon knapp är nertryckt och *mouseDragged* anropas vid rörelser när någon knapp är det. I båda fallen vill vi uppdatera verktygets position så lägg till:

```
if(activeTool != null)
{
          activeTool.setX(e.getX());
          activeTool.setY(e.getY());
}
```

För *mouseDragged* vill vi också säga till verktyget att en knapp är nertryckt så anropa också *activeTool.mouseDragged*() efter *setY*. Vi vill också meddela när vänstra knappen tryckts ner och när den släppts upp igen. Vi skulle kunna skriva *mousePressed* så här:

```
if(e.getButton() == MouseEvent.BUTTON1 && activeTool != null)
          activeTool.mouseDown();
```

och mouseReleased så här:

```
if(e.getButton() == MouseEvent.BUTTON1 && activeTool != null)
          activeTool.mouseUp();
```

Vi är färdiga med vyns hantering av verktyg men än så länge har vi inga verktyg definierade! Låt oss fixa det. Vi skapar en enkel penna, gör en ny klass som ärver av *MyTool* och kalla den för, säg, *MyPen*. Definiera en konstruktor som tar emot samma argument som *MyTool* och passa vidare dem till via super-anropet. Exempelvis såhär:

```
public MyPen(MyModel myModel, int x, int y, int size,
                                        Color color)
{
          super(myModel, x, y, size, color);
}
```

Överlagra därefter de abstrakta metoderna från *MyTool*. Vi börjar med att göra en (mycket) enkel ikon för vår penna. Lägg till en *BufferedImage* som en privat medlemsvariabel och instansiera den i konstruktorn, säg 32x32 pixlar. Använd objektets *getGraphics* och färga hela bilden gul (eller någon annan färg). I den överlagrade *getIcon* returnerar du ikonvariabeln. Då ska vi bara fixa de tre övriga metoderna i klassen. I vår enkla penna behöver i inte bry oss om *mouseUp* så den kan vara tom. Men på en position där vi klicka ner musknappen och en position dit vi dragit musen vill vi rita. Vi kan använda samma kod för båda, här är ett förslag:

```
Graphics2D g = getMyModel().getModelModifier();
g.setColor(getColor());
g.setStroke(new BasicStroke(getSize()));
g.drawRect(getX(), getY(), 1, 1);
```

För att testa om vår penna fungerar så lägger vi till en statisk penna till *MyView*s konstruktor:

```
tools.add(new MyPen(myModel, 50, 50, 3, Color.black));
```

Testa!

8. Observerare

Ok, det var inte helt lyckat, eller hur? Komponenten ritas inte om när vi börjar interagera med den, vi måste minimera och återställa fönstret för att se vad som händer. Hur kan vi då lösa detta? Jo, en första enkel grej vi kan göra är att lägga till ett anrop till *repaint*() när vi vet att komponenten inte stämmer längre, förslagsvis i *mousePressed*, *mouseReleased*, *mouseMoved* och *mouseDragged*. Men det här hjälper bara till hälften, om vi öppnar fler vyer så uppdaterar den bara sig själv. Låt oss därför att använda ett designmönster som heter *Observer* för att lösa detta. Mönstret finns färdigskrivet med hjälpklasser i Java, så låt oss använda dessa. Den första heter *Observable* och gör att andra kan *prenumerera* på uppdateringar från den här klassen. Låt vår modellklass *MyModel* ärva från *Observable*! Det är vår vy-komponent som beror på hur modellen ser ut, så låt den implementera interfacet *Observer*. Vi måste då implementera en metod *update* för att uppfylla interfacet *Observer*, så gör det. Den här metoden kommer att anropas när vår modell har förändrats, så lägg till en *repaint*() i den. Eftersom vi vill att våra *MyView*-objekt ska *prenumerera* på förändringar från *MyModel* så lägg till i konstruktorn:

```
myModel.addObserver(this);
```

Nu ska vi bara lägga till en upplysning om när vi har ändrat på modellen. I en enkel version av designmönstret skulle det räcka med att använda metoden *notifyObservers*() för att *update*-metoden hos alla *prenumeranter* ska anropas med i Java har de byggt in en mekanism som gör att man måste anropa *setChanged*() först innan det går. Metoden *setChanged* är tyvärr *protected* så vi kanske gör enklast i att skapa en ny metod till *MyModel* enligt:

```
public void informObservers()
{
        setChanged();
        notifyObservers();
}
```

Därefter kan vi anropa

```
getMyModel().informObservers();
```

i vår *Pen*-klass (och alla andra verktyg vi definierar) efter varje gång vi ritat på modellen. Testa igen så ser det nog bättre ut!

9. Koppla in menyn

Ta bort testpennan från *MyView*-konstruktorn, nu kopplar vi in den riktiga menyn istället. Så här skulle vi kunna lägga till en penna till verktygsmenyn:

```
JMenuItem toolPenItem = new JMenuItem("Penna");
newToolMenu.add(toolPenItem);
toolPenItem.addActionListener(this);
toolPenItem.setActionCommand("new_pen");
```

Vi lägger till en motsvarande händelse i *actionPerformed*(…):

```
...
else if(e.getActionCommand().equals("new_pen") &&
                        myDesktop.getSelectedFrame() != null)
{
                MyInternalFrame frame =
(MyInternalFrame)myDesktop.getSelectedFrame();
                frame.addTool(new MyPen(myModel, 24, 24, 1,
Color.black));
}
...
```

…fast *addTool* finns ju inte i *MyInternalFrame* utan i *MyView*, eller hur? Så skapa en sådan metod också i *MyInternalFrame* som bara passar vidare verktyget till vyn. Så att det nya verktyget ska synas när det läggs till, gör en *repaint*() i *MyView*-klassens *addTool*.

För att vi ska kunna ändra på ett aktivt verktyg måste vi kunna hämta det från huvudfönstret. Skapa därför en get-metod för *activeTool* i *MyView* och en metod med samma namn i *MyInternalFrame* som skickar vidare verktyget från vyn. Därefter kan vi skriva följande till *actionPerformed* i *MyFrame*:

```
...
else if(e.getActionCommand().equals("size") &&
                        myDesktop.getSelectedFrame() != null)
{
                JMenuItem menuItem = (JMenuItem)e.getSource();
                MyInternalFrame frame =

(MyInternalFrame)myDesktop.getSelectedFrame();
                MyTool tool = frame.getActiveTool();
                if(tool != null)

tool.setSize(Integer.parseInt(menuItem.getText()));

}
else if(e.getActionCommand().equals("color") &&
                        myDesktop.getSelectedFrame() != null)
{
                JMenuItem menuItem = (JMenuItem)e.getSource();
                MyInternalFrame frame =

(MyInternalFrame)myDesktop.getSelectedFrame();
                MyTool tool = frame.getActiveTool();
                if(tool != null)
                {
                        if(menuItem.getText().equals("Red"))
                                tool.setColor(Color.red);
                        if(menuItem.getText().equals("Blue"))
                                tool.setColor(Color.blue);
                        if(menuItem.getText().equals("Black"))
                                tool.setColor(Color.black);
```

```
                                if(menuItem.getText().equals("Green"))
                                          tool.setColor(Color.green);
                    }
          }
          ...
```

10. Resten

Nu är nästan allting färdigt, det behövs lite fler verktyg och andra små inställningar så programmet fungerar som du vill. Framför allt vill du nog fixa lite snyggare verktygsikoner. Klassen *ImageIO* kan du använda om du vill läsa in bilder från fil, annars är det inte särskilt svårt att rita en ikon med bara en Graphics2D. För ett exempel på ett lite mer avancerat (fast inte så mycket…) verktyg har jag bifogat koden för ett linjeverktyg. Att göra verktyg för elipser, rektanglar fungerar i princip på samma sätt. Prova gärna att göra verktyg som manipulerar bilden på lite ovanliga sätt, exempelvis ett inverteringsverktyg (inverterar pixelvärden där den ritar) eller ett blur-verktyg (jämnar ut pixelvärden där den ritar).

Bilaga A: MyLineTool.java

```
package grip08;       //Byt ut till ditt paketnamn!

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.image.BufferedImage;

public class MyLineTool extends MyTool
{
          private Image icon;
          private boolean isDrawing;
          private int mouseLastX;
          private int mouseLastY;

          public MyLineTool(MyModel myModel, int x, int y, int size,
                                                    Color color)
          {
                    super(myModel, x, y, size, color);
                    icon = new BufferedImage(32, 32,
                                        BufferedImage.TYPE_INT_ARGB);

                    Graphics g = icon.getGraphics();
                    g.setColor(Color.green);
                    g.fillRect(0, 0, 32, 32);

                    isDrawing = false;
                    mouseLastX = mouseLastY = -1;
          }

          @Override
          protected Image getIcon()
          {
                    return icon;
          }

          @Override
```

```
		public void mouseDown()
		{
			if(mouseLastX < 0 && mouseLastY < 0)
			{
				mouseLastX = getX();
				mouseLastY = getY();
				isDrawing = true;
			}
		}

		@Override
		public void mouseDragged()
		{
		}

		@Override
		public void mouseUp()
		{
			if(mouseLastX != -1 && mouseLastY != -1)
			{
				Graphics2D g =
					getMyModel().getModelModifier();
				g.setColor(getColor());
				g.setStroke(new BasicStroke(getSize()));

				g.drawLine(mouseLastX, mouseLastY,
						getX(), getY());

				isDrawing = false;
				mouseLastX = mouseLastY = -1;
			}
		}

		@Override
		public void paintToolOnSurface(Graphics2D g)
		{
			//Rita verktyget som vanligt
			super.paintToolOnSurface(g);

			//Rita vår temporära linje
			if(isDrawing)
			{
				g.setColor(getColor());
				g.setStroke(new BasicStroke(getSize()));

				g.drawLine(mouseLastX, mouseLastY,
						getX(), getY());
			}
		}
}
```