

Java-Simulation von digitalen Schaltungen

Inhalt:

1. Bezug zum Unterricht
2. Bezug zum bisherigen Material
3. Klassenhierarchien
4. Die Erzeugung von Komponenten unter Java
 - 4.1 Nands
 - 4.2 Schalter
5. Weitere Quelltexte
 - 5.1 Geräte
 - 5.2 Gatter
 - 5.3 Buchsen
 - 5.4 Der Simulator
6. Aufgaben

1. Bezug zum Unterricht

Die Simulation von digitalen Bausteinen wird im entsprechenden Abschnitt beschrieben, der allerdings Delphi-Programme benutzt. Hier sollen nur die Änderungen erwähnt werden, die aus der Nutzung des Java-Systems folgen – und da gibt es in der Tat einige:

- Man muss sich entscheiden, ob man eine **Anwendung** (mit den Vorteilen der Nutzung des Designers (in J++)) erstellen will, oder ein **Applet**.
- Applets haben den Charme, leicht zu **veröffentlichen** zu sein. Beim Speichern der Simulationsergebnisse wird es dann aber problematisch. Sie eignen sich also gut für erste Erfahrungen mit digitaler Elektronik – für ernsthafte Arbeit über mehrere Stunden sind sie weniger geeignet.
- Da Java nicht mehr benutzte Objekte selbst „abräumt“ (**garbage-collection**), brauchen wir uns darum im Programm nicht zu kümmern. Das ist für größere Projekte ein nicht zu unterschätzender Vorteil, denn viele Laufzeitfehler haben ihre Ursache gerade in einer fehlerhaften Speicherverwaltung.
- Weil Java **Threads** sehr elegant als Interfaces unterstützt, können wir diese Eigenschaft für simulierte Bauteile ausnutzen, die in der Realität autonom und parallel arbeiten.

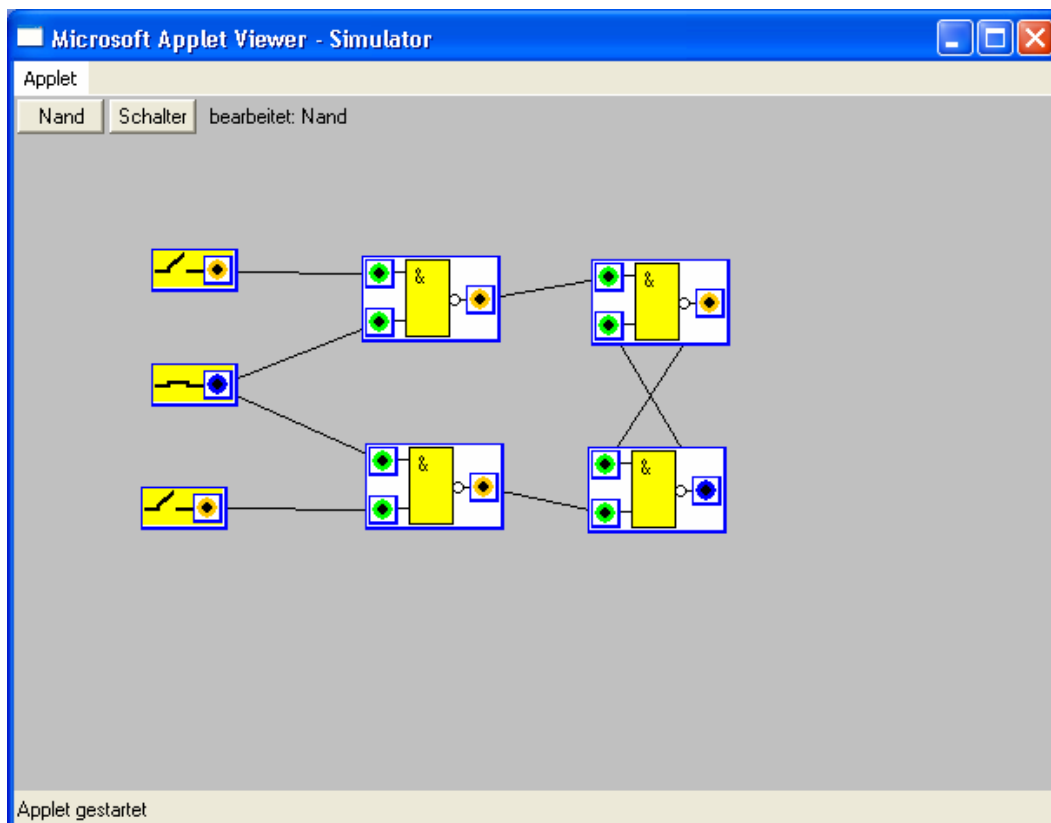
2. Bezug zum bisherigen Material

Unabhängig von anderen Überlegungen müssen wir in der Unterrichtsfolge auch auf eine gewisse Effizienz achten, also nicht immer neu das Rad erfinden (wenn es sich vermeiden lässt). Da wir in Kurs 2 bei der Einführung in die OOP schon *Physikgeräte* (→ *Kurs 2, Klassen und Objekte, Abschnitt 1.3*) kennen gelernt haben (welch ein Zufall!), wollen wir die dort eingeführten Methoden auf die Simulation von Gattern übertragen. Damit ist die Entscheidung gefallen – wir arbeiten mit Applets. Weiterhin wollen wir die Reduktionen des Simulators *Hasi* (→ *Kurs 3, Simulation von digitalen Schaltungen*) beibehalten. Weil sich die Ereignissteuerung in Java und Delphi unterscheidet – und aus Gründen der Einfachheit – realisieren wir hier nicht vollständig das bei Delphi benutzte UML-Diagramm, sondern benutzen die schon bekannten Klassen.

Das System soll also in einer sehr reduzierten, aber leicht erweiterbaren Form realisiert werden. Dazu sollen

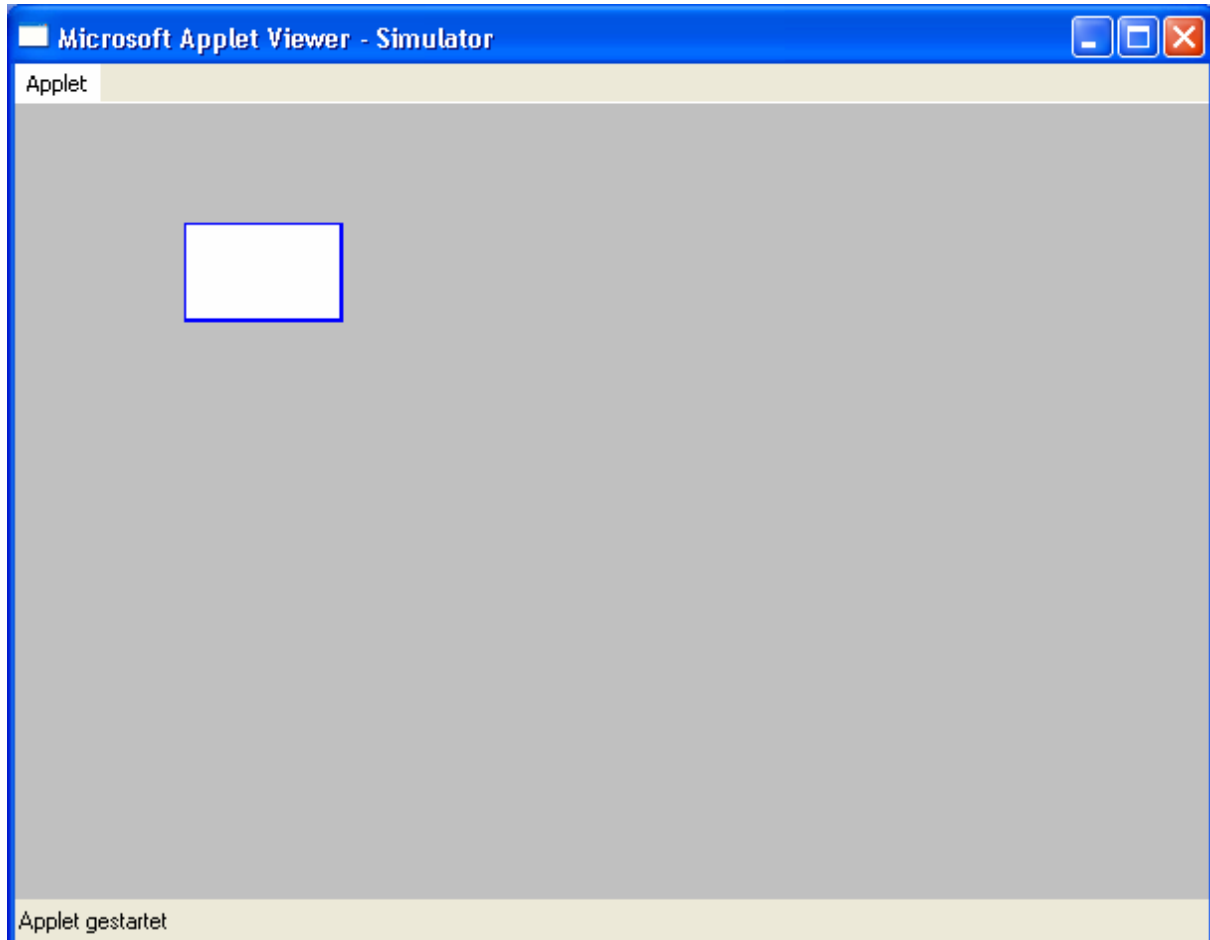
- nur zwei Bauteile erzeugbar sein: NANDs und SCHALTER
- die Bauteile am Bildschirm verschiebbar und bedienbar sein.
- eine halbwegs übersichtliche Objekthierarchie erzeugt werden, in die später weitere Komponenten leicht eingeklinkt werden können.
- die benötigten Leitungen nur sehr rudimentär angezeigt werden. (Die Verdrahtung der Komponenten ist ein eigenes, ziemlich komplexes Problem.)
- rein objektorientiert programmiert werden. D. h. die erzeugten Bausteine werden nicht vom Programm verwaltet, sondern arbeiten „autonom“ selbstständig vor sich hin. Die Aktionen werden vollständig ereignisgesteuert ausgelöst.

Als Programmiersprache wird Java verwendet.



3. Klassenhierarchien

Unsere Digitalbausteine sollen am Bildschirm verschiebbare Elemente sein, die auf Mausergebnisse reagieren. Wir kennen solche *Geraet*(e) schon von den Physikobjekten her. Dort wurden sie von *Panel*(s) abgeleitet und um die Möglichkeit erweitert, mithilfe der Maus verschiebbar zu sein. *Geraete* sind damit auch als Basisklasse für simulierte *Gatter* geeignet.



Aus *Geraeten* leiten wir dann die benutzten Klassen ab:

- nur zeitweise aktive Geräte wie *Schalter*, ..., die nur über Ausgänge verfügen, reagieren auf Ereignisse und brauchen keinen eigenen Thread.
- daueraktive Geräte, deren Zustand von ihren Eingangswerten abhängt und die ihre Eingänge deshalb dauern „befragen“ müssen, können dieses mithilfe eines eigenen Threads erledigen. Typische Beispiele hierfür sind eben die *Gatter*.

Wir taufen deshalb unsere Klasse *GeraeteMitThread* um in *Gatter*. Unsere *Buchse(n)*-Klasse behalten wir bei. Nur die verarbeiteten *Werte* ändern wir von *double* in *boolean* (weil wir es jetzt mit digitaler Elektronik zu tun haben). Aus *Gattern* und unter Benutzung von *Buchsen* werden dann die benötigten Spezialgatter wie z. B. *Nands mit zwei Eingängen* gebaut.

4. Die Erzeugung von Komponenten unter Java

Bei Anwendungen erzeugt man visuelle Komponenten (Buttons, ...) im Objektdesigner und stattet sie im Objektinspektor mit Eigenschaften aus. Stattdessen kann man Komponenten aber auch dynamisch während des Programmlaufs erzeugen. Dazu

- vereinbaren wir eine Komponente als Variable → ergibt einen leeren Zeiger
- rufen den entsprechenden Konstruktor auf → auf dem Heap wird Platz für die Attribute des Objekts belegt
- weisen den Attributen Werte zu → das Objekt erhält z. B. die richtige Größe und Farbe
- und übergeben die Komponente dem Applet (mit *add(...)*).

Als Beispiel sehen wir uns die Erzeugung eines Nands an:

```
import ...

public class Simulator extends Applet implements MouseListener, ...
{
    Nand N;
    String Bauteil = "Nand";
    ...

    public void init()
    {
        addMouseListener(this);
        ...
    }

    public void stop()
    {...}

    public void mouseReleased(MouseEvent e)
    {
        if(Bauteil=="Nand")
        {
            N = new Nand(e.getX(),e.getY());
            N.werdeVerschiebbar(); // wg. der Kompatibilität zu den Physikgeräten, sonst überflüssig
            add(N);
        }
        ...
    }
}
```

Nach seiner Erzeugung „lebt“ das Objekt in der Windowswelt, reagiert auf Mausereignisse und ist – sobald ein weiteres Objekt erzeugt wurde – der Kontrolle durch unser Applet entzogen, weil wir (hier) keinen besonderen Verweis auf das Objekt speichern. Wir vergessen das Ding einfach auf dem Bildschirm.

4.1 Nands

Bei einem NAND handelt es sich um ein Gatter mit zwei Eingängen und einem Ausgang sowie einem Schaltsymbol auf dem *Bild* des Gatters, das ein Nand darstellt.

```
import java.awt.*;

public class Nand extends Gatter
{
    Buchse e1, e2, a;
```

Bei der Erzeugung eines Nands müssen die entsprechenden Buchsen an den richtigen Stellen platziert werden.

```
public Nand(int x, int y)
{
    super(x, y, 80, 50, Color.white);
    a = new Buchse(60, 17, Color.red, "Ausgang", this);
    add(a);
    e1 = new Buchse(2, 2, Color.green, "Eingang", this);
    add(e1);
    e2 = new Buchse(2, 30, Color.green, "Eingang", this);
    add(e2);
}
```

Danach wird das Schaltsymbol gezeichnet.

```
public void paint(Graphics g)
{
    super.paint(g); //Das leere Gatter zeichnen
    g.setColor(Color.black); //... danach Linien usw.
    g.drawLine(18, 9, 25, 9);
    g.drawLine(18, 37, 25, 37);
    g.drawLine(56, 25, 60, 25);
    g.drawRect(25, 2, 25, 44);
    g.drawOval(50, 22, 6, 6);
    g.setColor(Color.yellow); //... und jetzt ein gelbes Rechteck wg. der Schönheit
    g.fillRect(26, 3, 24, 43);
    g.setColor(Color.black);
    g.drawString("&", 30, 16);
}
```

Die Methode *arbeite()* wird periodisch vom Thread aufgerufen. Hier wird aus den Eingangsbelegungen der Wert des Ausgangs bestimmt: $a = e_1 \wedge e_2$.

```
public void arbeite()
{
    a.wert = !(e1.wert() && e2.wert());
    a.repaint();
}
}
```

.... fertig!

4.2 Schalter

Bei einem *Schalter* handelt es sich um ein Gerät mit einem Ausgang sowie einem Zustand, der angibt, ob der Schalter geöffnet oder geschlossen ist. (Geschlossene Schalter liefern am Ausgang den Wert 0.)

```
import java.awt.*;

public class Schalter extends Geraet
{
    Buchse a;
    boolean zustand = false;
```

Der Konstruktor erzeugt wie üblich die Buchse.

```
    public Schalter(int x,int y)
    {
        super(x,y,50,25,Color.white);
        a = new Buchse(30,4,Color.red,"Ausgang",this);
        add(a);
    }
```

Dann wird der Schalter gezeichnet – je nach Zustand mit offenem oder geschlossenem „Balken“.

```
    public void paint(Graphics g)
    {
        super.paint(g);
        g.setColor(Color.black);
        g.setColor(Color.yellow);
        g.fillRect(1,1,46,21);
        g.setColor(Color.black);
        g.drawLine(2,13,10,13);
        g.drawLine(2,12,10,12);
        g.drawLine(20,13,30,13);
        g.drawLine(20,12,30,12);
        if(!zustand)
        {
            g.drawLine(9,10,21,10); // geschlossen
            g.drawLine(9,11,21,11);
        }
        else
        {
            g.drawLine(9,10,18,2); // offen
            g.drawLine(9,11,18,3);
            g.drawLine(9,12,18,4);
        }
    }
}
```

Mausereignisse rufen bei Geräten die Methode *verarbeite()* auf. Hier wird der Zustand des Schalters geändert.

```
    public void verarbeite(int x, int y)
    {
        zustand = !zustand;
        a.wert = zustand;
        repaint();
    }
}
```

.... fertig!

5. Die restlichen Quelltexte

Da auch unsere Physikgeräte schon Buchsen hatten, die mit einander verbunden werden konnten, folgen jetzt der Vollständigkeit halber nur noch die schon dort erläuterten reinen Quelltexte.

5.1 Geräte

```
import java.awt.*;
import java.awt.event.*;

public class Geraet extends Panel
    implements MouseMotionListener,MouseListener
{
    int x,y,b,h;
    Color hgf=Color.white,rf=Color.blue,sf=Color.black; //Farben
    static Graphics Bild; //Klassenfeld
    int xOffset,yOffset; // für Mausarbeiten
    boolean sichtbar,verschiebbar=false;

    static Buchse Eingangsbuchse=null, Ausgangsbuchse=null;

    public void setColor(Color c)
    {
        hgf = c;
    }

    public Geraet() //default-Konstruktor
    {
        this(100,100,80,50,Color.white);
    }

    public Geraet(int x, int y, int b, int h) //überladene Konstruktoren
    {
        this(x,y,b,h,Color.white);
    }

    public Geraet(int x, int y, int b, int h, Color hgf)
    {
        this.x = x;
        this.y = y;
        this.b = b;
        this.h = h;
        setLayout(null);
        setBounds(x,y,b,h);
        setBackground(hgf);
        setForeground(sf);
        addMouseListener(this);
    }

    public void werdeVerschiebbar()
    {
        verschiebbar = true;
        addMouseMotionListener(this);
    }

    public void verarbeite(int x, int y)
    {
        //die Methode wird von „echten“ Geräten ersetzt
    }
}
```

```
public void verbinde()
{
}

public void paint(Graphics g)
{
    g.setColor(rf);
    g.drawRect(0,0,b-1,h-1);
    g.drawLine(0,h-2,b-1,h-2);
    g.drawLine(b-2,0,b-2,h-1);
}

public void mousePressed(MouseEvent e)
{
    if(verschiebbar)
    {
        int x = e.getX(), y = e.getY();
        xOffset = x-this.x;
        yOffset = y-this.y;
        setVisible(false);
        Bild.setXORMode(Color.white);
        Bild.setColor(Color.black);
        sichtbar = false;
    }
}

public void mouseDragged(MouseEvent e)
{
    if(verschiebbar)
    {
        int x = e.getX(), y = e.getY();
        if(sichtbar)
        {
            Bild.drawRect(this.x,this.y,b,h);
            sichtbar = false;
        }
        else
        {
            this.x = x - xOffset;
            this.y = y - yOffset;
            Bild.drawRect(this.x,this.y,b,h);
            sichtbar = true;
        }
    }
}

public void mouseReleased(MouseEvent e)
{
    int x = e.getX(), y = e.getY();
    if(verschiebbar)
    {
        if(sichtbar) Bild.drawRect(this.x,this.y,b,h);
        this.x = x - xOffset;
        this.y = y - yOffset;
        setBounds(this.x,this.y,b,h);
        Bild.setPaintMode();
        setVisible(true);
    }
    else verbinde();
}
```

```
public void mouseClicked(MouseEvent e)
{
    verarbeite(e.getX(), e.getY());
}

public void mouseEntered(MouseEvent e)
{
}

public void mouseExited(MouseEvent e)
{
}

public void mouseMoved(MouseEvent e)
{
}
}
```

5.2 Gatter

```
import java.awt.*;
import java.awt.event.*;

public class Gatter extends Geraet implements Runnable
{
    Thread t; //für die kontinuierliche Arbeit

    public void stop()
    {
        t.interrupt();
    }

    public Gatter() //default-Konstruktor
    {
        this(100,100,80,50,Color.white);
    }

    public Gatter(int x, int y, int b, int h) //überladene Konstruktoren
    {
        this(x,y,b,h,Color.white);
    }

    public Gatter(int x, int y, int b, int h, Color hgf)
    {
        super(x,y,b,h,hgf);
        t = new Thread(this);
        t.start();
    }

    public void arbeite()
    {
        //die Methode wird von „echten“ Geräten ersetzt
    }

    public void run()
    {
        while(true)
        {
            if(t.isInterrupted()) break;
            arbeite();
            try
            {
                t.sleep(10);
            }
            catch(InterruptedException e){}
        }
    }
}
```

5.3 Buchsen

```
import java.awt.*;

public class Buchse extends Geraet
{
    Buchse kontakt=null;
    boolean wert=true; // unbelegte Eingänge liegen auf „1“
    String typ="Ausgang";
    Geraet besitzer;

    public Buchse(int x,int y, Color c, String t, Geraet owner)
    {
        super(x,y,17,17,new Color(200,200,200));
        sf = c;
        if(t.equals("Eingang")) typ = "Eingang"; else typ = "Ausgang";
        besitzer = owner;
    }

    public void paint(Graphics g)
    {
        super.paint(g);
        if(typ.equals("Eingang")) g.setColor(Color.green);
        else if(wert==false) g.setColor(Color.blue);
        else g.setColor(Color.orange);
        g.fillOval(2,2,12,12);
        g.setColor(Color.black);
        g.fillOval(5,5,6,6);
    }

    public void setzeVerbindung(Buchse b)
    {
        if(typ.equals("Eingang")) kontakt = b;
        else kontakt = null;
    }

    public boolean wert()
    {
        if(typ.equals("Eingang"))
            if(kontakt==null) return true; // unbelegte Eingänge liegen auf „1“
            else return kontakt.wert();
        else return wert;
    }

    public void verbinde()
    {
        int xanf=0,yanf=0,xend=0,yend=0;
        if(typ.equals("Eingang"))
        {
            if(Ausgangsbuchse!=null)
            {
                setzeVerbindung(Ausgangsbuchse);
                Ausgangsbuchse.setBackground(Ausgangsbuchse.hgf);
                Ausgangsbuchse.repaint();
                setBackground(hgf);
                repaint();
                Bild.setPaintMode();
                Bild.setColor(Color.black);
                xanf = Ausgangsbuchse.besitzer.x+Ausgangsbuchse.x+8;
                yanf = Ausgangsbuchse.besitzer.y+Ausgangsbuchse.y+8;
                xend = besitzer.x+x+8;
            }
        }
    }
}
```

```
        yend = besitzer.y+y+8;
        Bild.drawLine(xanf, yanf, xend, yend);
        Ausgangsbuchse = null;
        Eingangsbuchse = null;
    }
    else
    {
        Eingangsbuchse = this;
        setBackground(Color.blue);
        repaint();
    }
}
else
{
    if (Eingangsbuchse != null)
    {
        Eingangsbuchse.setzeVerbindung(this);
        Eingangsbuchse.setBackground(Eingangsbuchse.hgf);
        Eingangsbuchse.repaint();
        setBackground(hgf);
        repaint();
        Bild.setPaintMode();
        Bild.setColor(Color.black);
        xanf = besitzer.x+x+8;
        yanf = besitzer.y+y+8;
        xend = Eingangsbuchse.besitzer.x+Eingangsbuchse.x+8;
        yend = Eingangsbuchse.besitzer.y+Eingangsbuchse.y+8;
        Bild.drawLine(xanf, yanf, xend, yend);
        Ausgangsbuchse = null;
        Eingangsbuchse = null;
    }
    else
    {
        Ausgangsbuchse = this;
        setBackground(Color.blue);
        repaint();
    }
}
}
}
```

5.4 Der Simulator

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class Simulator extends Applet
    implements ActionListener,MouseListener
{
    Nand N;      // Die Geräte kennt er nur!
    Schalter S;
    Button bNand    = new Button("Nand");// rudimentäre Steuerelemente (s. Screenshot vorne).
    Button bSchalter = new Button("Schalter");
    Label lAnzeige  = new Label("bearbeitet: Nand");
    String Bauteil  = "Nand";

    public void init()
    {
        setLayout(null); // „Oberfläche“ erzeugen
        bNand.setBounds(2,2,50,20);
        bNand.addActionListener(this);
        add(bNand);
        bSchalter.setBounds(55,2,50,20);
        bSchalter.addActionListener(this);
        add(bSchalter);
        lAnzeige.setBounds(110,2,200,20);
        add(lAnzeige);
        Geraet.Bild = getGraphics();
        addMouseListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        Bauteil = e.getActionCommand(); // Anzeige wechseln
        lAnzeige.setText("bearbeitet: "+Bauteil);
    }

    public void mousePressed(MouseEvent e) {}
    public void mouseDragged(MouseEvent e) {}

    public void mouseReleased(MouseEvent e)
    {
        if(Bauteil=="Nand") // hier werden Bauteile erzeugt
        {
            N = new Nand(e.getX(),e.getY());
            N.werdeVerschiebbar();
            add(N);
        }
        if(Bauteil=="Schalter")
        {
            S = new Schalter(e.getX(),e.getY());
            S.werdeVerschiebbar();
            add(S);
        }
    }

    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseMoved(MouseEvent e) {}
}
```

8. Aufgaben

Ergänzen Sie den Hardwaresimulator wie folgt:

1. Fügen Sie weitere Grundgatter mit zwei Eingängen und einem Ausgang ein: *UND2*, *ODER2*, *EXOR*
2. Führen Sie einen *NICHT*-Baustein ein, der direkt aus der Gatter-Klasse abgeleitet wird.
3. Führen Sie eine Klasse *Gatter4E* der Gatter mit vier Eingängen ein. Leiten Sie aus dieser entsprechende Grundgatter ab.
4. Führen Sie Rechenschaltungen ein: Halbaddierer *HA* und Volladdierer *VA*.
5. Entwickeln Sie ein *Auffang-Flipflop*, also eine Schaltung, die ein Bit speichern kann.
6. Leiten Sie aus dem Auffangflipflop ein *JK-MS-FF* ab.
7. Entwickeln Sie *Binärzähler*, *Register* und ein *RAM*.
8. a: Versuchen Sie, die Probleme beim Einsatz etwas übersichtlicher *Leitungen* im Simulator einzuschätzen. Diskutieren Sie verschiedene Möglichkeiten und den zu deren Lösung erforderlichen Aufwand.
b: Leiten Sie aus der Bausteinklasse eine Klasse *Leitungsknoten* ab, zwischen denen Leitungsstücke verlaufen. Machen Sie die Knoten bei Bedarf verschiebbar.
c: Führen Sie neue Knotentypen *Verzweigung* ein, von dem aus Leitungen in zwei bzw. drei Zweige aufgespaltet werden können.
9. Diskutieren Sie den Aufwand, der bei der Umstellung des Simulators auf eine *Time-Priority-Queue* anfällt. Diese Schlange verwaltet den einzigen Timer des Systems, in dessen Warteschlange sich Bausteine eintragen und von dem nach Verlauf eines entsprechenden Zeitintervalls die *arbeite*-Methode der Bausteine aufgerufen werden.