

# **Einführung in die Informatik - Teil XI - Java-Simulation von digitalen Schaltungen**

## **Inhalt:**

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

## 1. Bezug zum Unterricht

Für die Simulation von digitalen Bausteinen muss man einige Vorab-Entscheidungen treffen, die Konsequenzen für die gesamte Arbeit haben:

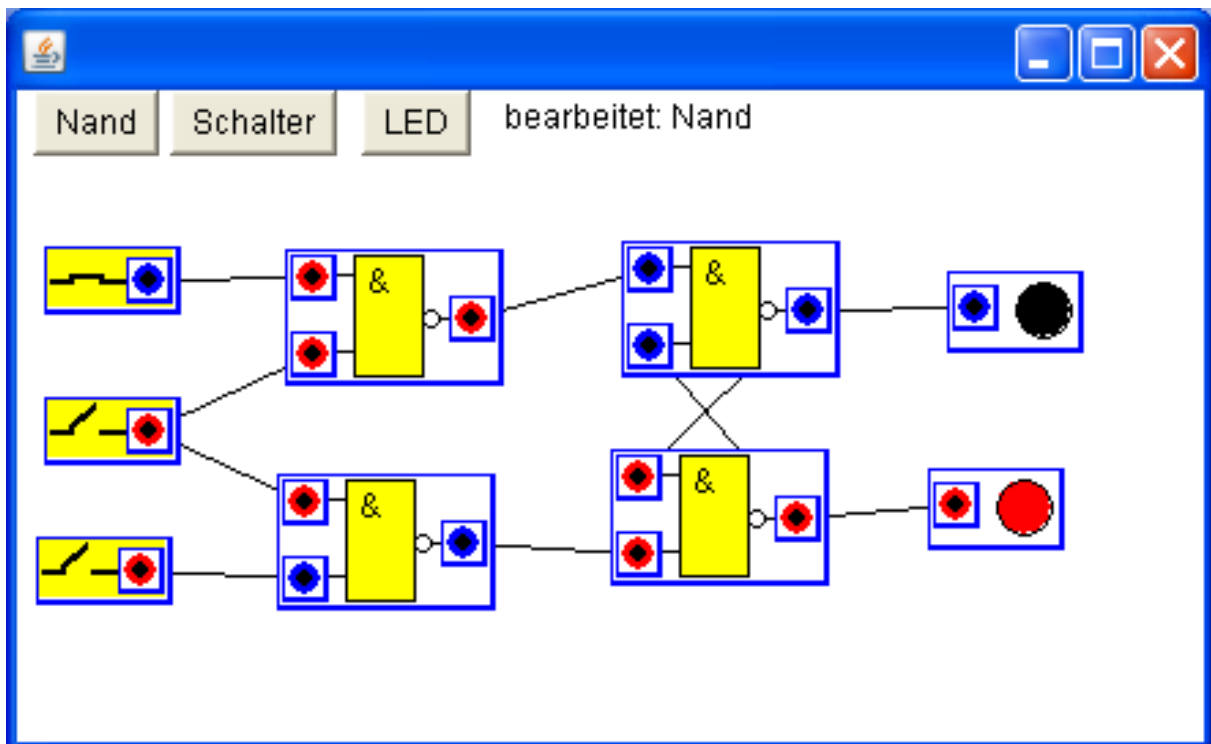
- Man muss sich entscheiden, ob man eine *Anwendung* (mit vollem Zugriff auf den Rechner) erstellen will, oder ein *Applet*. Applets haben den Charme, leicht zu *veröffentlichen* zu sein. Beim Speichern der Simulationsergebnisse wird es dann aber problematisch. Sie eignen sich also gut für erste Erfahrungen mit digitaler Elektronik – für ernsthafte Arbeit über mehrere Stunden eher weniger.
- 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. Einschränkungen

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

- Anwendungen realisiert werden,
- nur drei Bauteile erzeugbar sein: LEDs, 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, als Entwicklungssystem NetBeans 6.1.

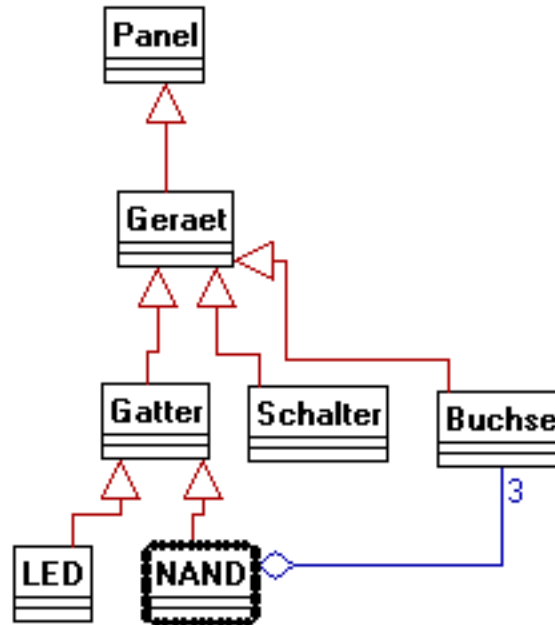


### 3. Klassenhierarchien

Unsere Digitalbausteine sollen am Bildschirm verschiebbare Elemente sein, die auf Mausereignisse reagieren. Wir nennen sie *Geraete*. Wir leiten sie von Panels ab und erweitern sie um die Möglichkeit, 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. Ähnlich sieht es mit *Buchsen* aus, die allerdings auf den Bausteinen nicht verschiebbar sein dürfen.
- 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 die *Gatter*.
- Von diesen werden konkrete digitale Bausteile abgeleitet wie NANDs, LEDs und weitere.



Als Beispiel enthält ein NAND drei Buchsen: zwei Eingänge und einen Ausgang.

## 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 Anwendungsframe (mit *add(...)*).
- 
- Als Beispiel sehen wir uns die Erzeugung eines NANDs an: Enthält die String-Variablen *Bauteil* den Text „Nand“, dann wird ein neues Nand an der Stelle erzeugt, an der die Maustaste losgelassen wurde, und dem Frame hinzugefügt.

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

public class FrameSchaltungssimulation extends java.awt.Frame
    implements MouseListener
{
    Nand N;            // Die Geräte kennt nur der Frame!
    String Bauteil = "Nand";

    ... //Erzeugung der Buttons etc

    public void mouseReleased(MouseEvent e)
    {
        // hier werden Bauteile erzeugt
        if (Bauteil.equals("Nand"))
        {
            N = new Nand(e.getX(), e.getY());
            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 Programm 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 wird zuerst der übergeordnete Konstruktor aufgerufen (der der Gatter-Klasse). Danach müssen die Buchsen an den richtigen Stellen platziert werden. Da NANDs am Bildschirm verschiebbar sein sollen, wird das ebenfalls eingestellt.

```
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);
    werdeVerschiebbar();
}
```

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:  $a = e_1 \wedge e_2$ . Da eventuell angeschlossene Buchsen schon wieder gelöscht wurden oder andere Katastrophen eintreten können, wird das Ganze in einen *try...catch...*-Block verpackt.

```
public void arbeite()
{
    try
    {
        a.wert = !(e1.wert() && e2.wert());
    } catch (Exception e)
    { a.wert = true; }
    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 = true;
}
```

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.BLUE, "Ausgang", this);
    werdeVerschiebbar(); //weil Schalter keine Gatter sind, extra angeben
    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.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!

### 4.3 LEDs

Bei einer **LED** handelt es sich um ein Gerät mit einem Eingang und einem Anzeigeelement, das seine Farbe in Abhängigkeit vom Eingangswert ändert. Da sich dieser Eingang zu unbekannten Zeitpunkten ändern kann, muss die LED dauernd arbeiten, also in regelmäßigen Abständen den Eingangswert abfragen. Sie wird deshalb aus der Gatterklasse abgeleitet.

```
import java.awt.*;

public class LED extends Gatter
{
    Buchse ein;
    Color farbe = Color.BLACK;
```

Der Konstruktor erzeugt wie üblich die Eingangsbuchse.

```
public LED(int x, int y)
{
    super(x, y, 50, 30, Color.WHITE);
    ein = new Buchse(2, 5, Color.RED, "Eingang", this);
    add(ein);
}
```

Beim Zeichnen wird der Zustand des Eingangs abgefragt und die „LED“ entsprechend gezeichnet.

```
public void paint(Graphics g)
{
    super.paint(g); //Das leere Gatter zeichnen
    try
    {
        if (ein.wert()) farbe = Color.RED;
        else farbe = Color.BLACK;
    }
    catch(Exception e){farbe = Color.BLACK;}
    ein.repaint();
    g.setColor(farbe);
    g.fillOval(25, 4, 20, 20);
    g.setColor(Color.BLACK);
    g.drawOval(25, 4, 20, 20);
}
```

Die regelmäßig vom Thread aufgerufene **arbeite**-Methode bewirkt nur ein Neuzeichnen.

```
public void arbeite()
{
    this.repaint();
}
}
```

.... fertig!

## 5. Die restlichen Quelltexte

Die eigentliche Arbeit liegt natürlich in den Basisklassen von Gattern, Schaltern und Buchsen versteckt. Diese heißen *Geraet* und *Gatter* (s. UML-Klassendiagramm).

### 5.1 Geräte

Geräte stellen die Grundfunktionalität bereit. Sie gestatten es, Rechtecke zu zeichnen, die den äußeren Rand der Geräte darstellen. Die Mausereignisse werden ausgenutzt, um einerseits die Geräte am Bildschirm verschieben zu können, andererseits die *verarbeite*-Methode der abgeleiteten Klassen aufzurufen. Da nicht alle Geräte verschiebbar sein sollen (Buchsen z.B. nicht), wird eine *werdeVerschiebbar*-Methode bereitgestellt, die bei Bedarf aufzurufen ist. Die Geräte werden mit gedrückter Maustaste verschoben und durch Anklicken aktiviert.

Die Verbindung von Geräten erfolgt später über das Anklicken von Buchsen. Dies wird über zwei *Klassenfelder* namens *Eingangsbuchse* und *Ausgangsbuchse* ermöglicht. Wird später eine Buchse angeklickt, dann wird diesen Feldern die entsprechende Referenz zugewiesen (s. Buchsen). Haben beide einen vernünftigen Wert, dann werden die Buchsen verbunden, indem der Eingang eine Referenz auf den Ausgang erhält. Die Steuerung dieses Vorgangs obliegt dem Simulationsprogramm, das die Geräte-Klasse benutzt.

Es hat keinen Sinn, Exemplare der Geräte-Klasse zu erzeugen, da diese noch keine Funktionalität besitzen. Eigentlich hätte die Klasse deshalb als *abstrakte Klasse* implementiert werden sollen. Ich habe der Einfachheit halber hier darauf verzichtet.

Die Klasse verfügt über drei überladene Konstruktoren, um diese Möglichkeit einmal zu demonstrieren. Nötig ist das nicht.

```
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 Mausearbeiten
    boolean sichtbar, verschiebbar = false;
    static Buchse Eingangsbuchse = null, Ausgangsbuchse = null; //für Verbindungen

    public void setColor(Color c) //sollte klar sein
    {
        hgf = c;
    }

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

    public Geraet(int x, int y, int b, int h) //überladener Konstruktor
    {
        this(x, y, b, h, Color.white);
    }
}
```

```
public Geraet(int x, int y, int b, int h, Color hgf) // der eigentliche Konstruktor
{
    this.x = x; // Attributwerte setzen
    this.y = y;
    this.b = b;
    this.h = h;
    setLayout(null);
    setBounds(x, y, b, h);
    setBackground(hgf);
    setForeground(sf);
    addMouseListener(this); // auf Mausklicks reagieren sie alle
}

public void werdeVerschiebbar() // ggf. auch auf Mausbewegungen reagieren
{
    verschiebbar = true;
    addMouseMotionListener(this);
}

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

public void verbinde()
{
    // für die Buchsen
}

public void paint(Graphics g) // den Rahmen zeichnen
{
    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) // Anfang des Verschiebungsprozesses
{
    if (verschiebbar) // z. B. bei Buchsen klappt es nicht
    {
        int x = e.getX(), y = e.getY(); // Mausklickposition merken
        xOffset = x - this.x;
        yOffset = y - this.y;
        setVisible(false); // nur den äußeren Rahmen verschieben (aus Zeitgründen)
        Bild.setXORMode(Color.WHITE); // Hintergrund erhalten
        Bild.setColor(Color.BLACK);
        sichtbar = false;
    }
}

public void mouseDragged(MouseEvent e) // sollte klar sein
{
    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) //jetzt das Gerat ggf. wieder vollstandig zeichnen
{
    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(); // nur bei Buchsen
    }
}

public void mouseClicked(MouseEvent e) // sollte klar sein (s.o.)
{
    verarbeite(e.getX(), e.getY());
}

// leere Methoden, die vorhanden sein mussen, um die Schnittstelle zu implementieren

public void mouseEntered(MouseEvent e)
{
}

public void mouseExited(MouseEvent e)
{
}

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

## 5.2 Gatter

Mit den Gattern ist es etwas einfacher, weil die fast alles von den Geräten erben. Neu hinzu kommt ein *Thread*, der hier als *Interface* implementiert wird.

```
import java.awt.*;

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

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

    public void run() // die kontinuierlich aufgerufene run-Methode des Threads
    {
        while(true)
        {
            if(t.isInterrupted()) break; // ggf. abbrechen
            arbeite(); // hier erfolgt die eigentliche Arbeit
            try
            {
                t.sleep(10); // kleine Pause, sollte eigentlich gerätespezifisch sein
            }
            catch(InterruptedException e){}
        }
    }

    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);
        werdeVerschiebbar(); // alle Gatter sind verschiebbar
        t = new Thread(this); // neuen Thread erzeugen ...
        t.start(); // ... und starten
    }

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

### 5.3 Buchsen

Buchsen können entweder Eingänge oder Ausgänge sein. Das entsprechende Attribut wird hier einfach als String gespeichert. Da sie nur auf Geräten Sinn machen, haben sie einen *Besitzer*, so dass sie den bei Bedarf ansprechen können. Ausgänge erhalten ihren Wert vom Besitzer zugewiesen, Eingänge haben entweder den Wert „1“ (oder „true“), weil bei TTL-Bausteinen nicht verbundene Eingänge auf logisch „1“ liegen, oder sie erhalten den Wert des verbundenen Ausganges. Werden Buchsen verbunden, dann erhält das Feld *kontakt* des Eingangs eine Referenz auf den verbundenen Ausgang.

Kompliziert ist eigentlich nur das Zeichnen. Damit Buchsen Verbindungsleitungen (einfache Linien) zeichnen können, enthält die Geräteklasse ein Klassenattribut *bild*, das eine Referenz auf den Grafikkontext des Simulators (Applet oder Frame) enthält. Auf diesem kann gezeichnet werden.

```
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) //Konstruktor
    {
        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) // Buchsen zeigen ihren Wert durch die Farbe an
    {
        super.paint(g);
        if(wert()== false) g.setColor(Color.BLUE);
        else g.setColor(Color.RED);
        g.fillOval(2,2,12,12);
        g.setColor(Color.BLACK);
        g.fillOval(5,5,6,6);
    }

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

    public boolean wert() //Ausgänge haben einen Wert, Eingänge ermitteln diesen
    {
        try
        {
            if(typ.equals("Eingang"))
                if(kontakt==null) return true; // unbelegte Eingänge liegen auf „1“
                else return kontakt.wert();
            else return wert;
        }
        catch(Exception e){return true;} //sicherheitshalber
    }
}
```

```
public void verbinde() // Das ist der komplizierteste Teil
{
    int xanf=0,yanf=0,xend=0,yend=0;
    if(typ.equals("Eingang"))
    {
        if(Ausgangsbuchse!=null) // Eingang mit Ausgang verbinden
        {
            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; //Eingangsbuchse merken
            setBackground(Color.blue);
            repaint();
        }
    }
    else
    {
        if (Eingangsbuchse != null) //Wiederum Eingang mit Ausgang verbinden
        {
            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 //Ausgang merken
        {
            Ausgangsbuchse = this;
            setBackground(Color.BLUE);
            repaint();
        }
    }
}
}
```

## 5.4 Der Simulator

Beim Simulator handelt es sich um eine einfache Anwendung mit einem Frame, der den Grafikkontext für die Bausteine liefert. Vorhanden sind drei Buttons, mit deren Hilfe man zwischen der Erzeugung der drei Bausteintypen Schalter, Nand und LED umschalten kann. Wird einer der Knöpfe gedrückt, dann merkt sich das System dieses in der Variablen Bauteil. Je nach eingestelltem Wert werden entsprechende Komponenten erzeugt. Voreingestellt ist ein Nand. Die Bauteile sind nach ihrer Erzeugung nicht mehr erreichbar, sie arbeiten autonom.

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

public class FrameSchaltungssimulation extends java.awt.Frame
    implements MouseListener
{
    Nand N;          // Die Geräte kennt nur der Simulator!
    Schalter S;
    LED led;
    String Bauteil = "Nand";

    //Automatisch erzeugter Quelltext
    //-----
    /** Creates new form FrameSchaltungssimulation */
    public FrameSchaltungssimulation()
    {
        initComponents();
        init();
    }
    // <editor-fold defaultstate="collapsed" desc="Generated Code">

    private void initComponents() {

        bNand = new java.awt.Button(); // die Buttons
        bSchalter = new java.awt.Button();
        lAnzeige = new java.awt.Label();
        bLED = new java.awt.Button();

        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent evt) {
                exitForm(evt);
            }
        });
        setLayout(null);

        bNand.setLabel("Nand");
        bNand.addMouseListener(new java.awt.event.MouseAdapter() {
            public void mouseClicked(java.awt.event.MouseEvent evt) {
                bNandMouseClicked(evt);
            }
        });
        bNand.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                bNandActionPerformed(evt);
            }
        });
        add(bNand);
        bNand.setBounds(10, 30, 46, 24);
```

```
bSchalter.setLabel("Schalter");
bSchalter.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        bSchalterActionPerformed(evt);
    }
});
add(bSchalter);
bSchalter.setBounds(60, 30, 61, 24);

lAnzeige.setText("bearbeitet: Nand");
add(lAnzeige);
lAnzeige.setBounds(180, 30, 200, 20);

bLED.setLabel("LED");
bLED.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        bLEDActionPerformed(evt);
    }
});
add(bLED);
bLED.setBounds(130, 30, 40, 24);

pack();
} // </editor-fold>

/** Exit the Application */
private void exitForm(java.awt.event.WindowEvent evt) {
    System.exit(0);
}

public static void main(String args[])
{
    java.awt.EventQueue.invokeLater(new Runnable()
    {
        public void run()
        {
            new FrameSchaltungssimulation().setVisible(true);
        }
    });
}

// Variables declaration - do not modify
private java.awt.Button bLED;
private java.awt.Button bNand;
private java.awt.Button bSchalter;
private java.awt.Label lAnzeige;
// End of variables declaration
//-----

public void init()
{
    setSize(800, 600);
    Geraet.Bild = getGraphics(); //Grafikkontext merken
    addMouseListener(this);
}
```

```
// Nach Buttonklick merken, welcher Bauteiltyp ab jetzt erzeugt wird

private void bNandActionPerformed(java.awt.event.ActionEvent evt)
{
    Bauteil = "Nand";
    lAnzeige.setText("bearbeitet: " + Bauteil);
}

private void bSchalterActionPerformed(java.awt.event.ActionEvent evt)
{
    Bauteil = "Schalter";
    lAnzeige.setText("bearbeitet: " + Bauteil);
}

private void bLEDActionPerformed(java.awt.event.ActionEvent evt)
{
    Bauteil = "LED";
    lAnzeige.setText("bearbeitet: " + Bauteil);
}

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

public void mousePressed(MouseEvent e) { } // Leere Methoden für die Interfaces
public void mouseDragged(MouseEvent e) { }
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.